REFERENCE MODEL BASED HIGH FIDELITY SIMULATION MODELING FOR MANUFACTURING SYSTEMS

A Dissertation
Presented to
The Academic Faculty

by

HANSOO KIM

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in the
School of Industrial and Systems Engineering

Georgia Institute of Technology
May 2004

REFERENCE MODEL BASED HIGH FIDELITY SIMULATION MODELING FOR MANUFACTURING SYSTEMS

Approved by:

Dr. Chen Zhou, Co-Chair

Dr. Leon F. McGinnis, Co-Chair

Dr. Christos Alexopoulos

Dr. Douglas A. Bodner

Dr. Sridhar Narasimhan

Date Approved: April 1, 2004

DEDICATION

To my family, for their love, prayer, and patience…

"Fear thou not; for I am with thee: be not dismayed; for I am thy God:
I will strengthen thee; yea, I will help thee;
yea, I will uphold thee with the right hand of my righteousness"
(Isaiah 41:10)

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Chen Zhou, for his friendly guidance, support, and encouragement throughout this study. Special thanks are due to co-advisor, Dr. Leon F. McGinnis, who has showed me the right attitude to research–for his precious insight, encouragement, and financial support. I would also like to thank Dr. Christos Alexopoulos, Dr. Douglas A. Bodner, and Dr. Sridhar Narasimhan for serving on the committee and offering constructive comments.

It is obvious that, without many contributions by a number of friends, my long journey at Georgia Tech would have been much harder and lonelier. I would like to thank all student members in The Keck Virtual Factory Lab, and especially Jin Young Choi who has been a wonderful friend to me. I am also grateful to all church members in Zion KUMC for their prayers and love.

Also, I must thank my parents and parents-in-law back in Korea. I know how much they have prayed for me with love and all the sacrifices they have made. My wife, Sook, surely deserves to receive my special thanks. Without her patience, prayer, and unselfish devotion, it would not have been possible to get through all the obstacles and face all the vicissitudes that I have met during my study. I also thank to my son, Joshua (Sunoo) who gave me great pleasure and kept me going with his precious love and laughter.

Above all, I truly thank God, who has always been with me and guided my steps in the path of His righteousness.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Today, discrete event simulation is the only reliable tool for detailed analysis of complex behaviors of modern manufacturing systems. However, building high fidelity simulation models is expensive. Hence, it is important to improve the simulation modeling productivity. In this research, we explore two approaches for the improvement of simulation modeling productivity. One approach is the Virtual Factory Approach, using a general-purpose model for a system to achieve various simulation objectives with a single high fidelity model through abstraction. The other approach is the Reference Model Approach, which is to build fundamental building blocks for simulation models of any system in a domain with formal descriptions and domain knowledge. In the Virtual Factory Approach, the challenge is to show the validity of the methodology. We develop a formal framework for the relationships between higher fidelity and lower fidelity models, and provide justification that the models abstracted from a higher fidelity model are interchangeable with various abstract simulation models for a target system. For the Reference Model Approach, we attempt to overcome the weak points inherited from ad-hoc modeling and develop a formal reference model and a model generation procedure for discrete part manufacturing systems, which covers most modern manufacturing systems.

CHAPTER I

INTRODUCTION

1.1 Introduction

Discrete-event simulation has been widely used for evaluating design alternatives and performing "what-if" analysis in manufacturing systems. Since simulation is more flexible and less restrictive than analytical methods, it is a practical and unique tool to analyze the dynamic behaviors of manufacturing systems (Banks *et al*. 1996, Law and McComas 1998, Pegden *et al*. 1990). A simulation model is a descriptive model that imitates the behaviors of a real system, and simulation modeling is a process to build a model abstracting the nature of a real system. A real system can be viewed as the source of information for modeling, which has embedded logical rules governing system operations. Manufacturing systems have been one of the largest application areas of simulation modeling since their logical rules are better understood than other systems (Law and McComas 1998). In this thesis, we focus on simulation modeling for manufacturing systems.

Compared with analytical methods, simulation is expensive in achieving the correct model and in analyzing the output results. It is especially expensive to develop detailed models for dynamic behaviors of modern manufacturing systems. Often, the level of detail in simulation models is compromised due to budget and time constraints, and it perhaps explains why the validity of simulation procedures is questioned.

Therefore, an important issue in simulation is to improve the simulation modeling productivity. One simple way for improving simulation modeling productivity is to reuse existing simulation models. In theory, reuse can reduce the development time and costs, and improve the quality of new models. However, in practice, it has been difficult to reuse existing models for improving simulation modeling productivity in remarkable ways (Paul and Taylor 2002). As Overstreet and Nance (2002) have mentioned, the reuse of existing simulation models is one of the *Grand Challenges* in simulation.

One of the reasons model reuse has not been effective for improving modeling productivity is that the focus has been on reusing merely the program codes of simulation instead of the entire results of simulation modeling. Simulation modeling is a comprehensive process including activities from system analysis to simulation coding and output analysis. While the final version of simulation code may take only a small portion of the time in process, the iterative analysis and design of simulation models take much more time and effort. In addition, if we want to reuse existing simulation codes for new simulation modeling, we need additional tasks to analyze the existing codes as to whether or not they can fit well with the design concepts for the new simulation model. Even if the existing simulation codes are reusable, sometimes it can mean just saving the effort involved in retyping. Hence, simply reusing existing simulation codes may not have as much impact on improving the simulation modeling productivity as we expect.

Therefore, in order to improve the simulation modeling productivity effectively, we need a well-designed simulation modeling methodology to be able to reuse systematically not only the simulation codes, but also the other analysis results of simulation modeling.

1.2 Two Axes to Improve Simulation Modeling Productivity

By current conventions, we can define simulation modeling as a process to build a proper simulation model for a given system to achieve modeling purposes. In this framework, a perfect model for a particular target system with given modeling purposes would not be appropriate if we try to reuse the same for other systems and/or different modeling purposes. From this perspective, the scope of simulation modeling can be generalized to improve modeling productivity with two independent dimensions: the scope of purpose and the scope of system domain.

Scope of Purpose

Current simulation modeling is based on *Occam's* view of modeling, which is that a simulation model should be just sufficient to meet the purposes of modeling for a given system (Brooks and Tobias 1996). This view of modeling generally produces simulation models with less coding effort and better execution performance. However, this focus on model minimality often makes reuse of simulation modeling difficult.

One of the main goals of manufacturing simulation is to evaluate system performance for various design and control alternatives. The modeling purposes can be represented as a set of observable variables that model designers can observe from the simulation model.

Let us suppose that for a target system, a set of observable variables in one model includes all of the variables that are observable in other simulation models. Then, we can call the model that has the supreme set of observable variables a *general model*. The general model covers all purposes of other simulation models for the target system.

Since the general model has more observable variables than other models, it is a representative model to characterize a target system with the highest fidelity. In general, higher fidelity models can be converted efficiently to lower fidelity models with abstraction methods. However, the reverse is not true. We have room here for the improvement of modeling productivity. For a target system, if we can build a general model, to be referred to as a *virtual factory* in the later chapters, we can reuse the system knowledge embedded in the virtual factory to generate many simulation models for any observable variables of interest through simple abstractions.

Scope of System Domain

Another observation of conventional simulation modeling is that it focuses on a single target system. Hence, it becomes difficult to reuse a model that was structured and optimized for a specific system for even similar systems. In reality, many manufacturing systems share common features. If there is a simulation modeling methodology to accumulate and use the domain knowledge for all systems in a specific domain, instead of focusing on a single system, we can improve modeling productivity for any system in the domain.

1.3 Examples for New Simulation Modeling Approaches

The independent axes to improve the simulation modeling productivity introduced in the previous sections were abstract. In this section, we illustrate each concept through analogical examples, and identify research issues and tasks for implementation.

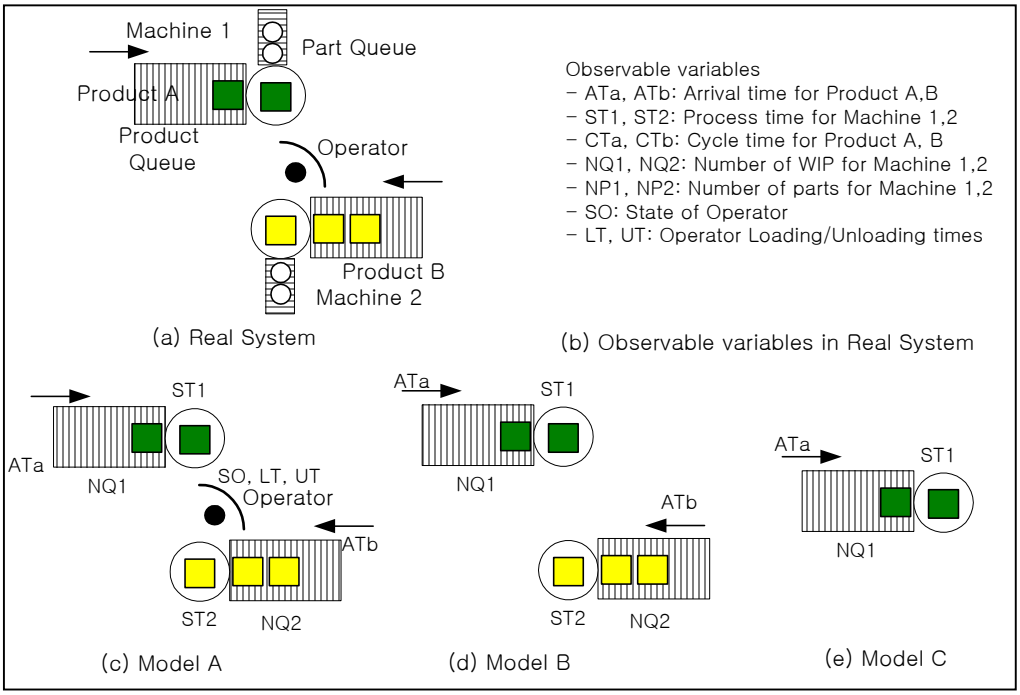


Figure 1.1 Two-Machine Manufacturing System and Its Models

Example I: Improving Productivity in the Dimension of Scope of Purpose

Figure 1.1 shows (a) a manufacturing system consisting of two machines with part queues and an operator for loading and unloading, (b) observable variables from the manufacturing system, and (c, d, e) three simulation models representing the manufacturing system.

This manufacturing system produces two types of product, Product A and B. Product A is processed on Machine 1 and Product B is processed on Machine 2. Products arrive and are stored in the respective queue until the machine becomes available. When a machine becomes available, the operator loads a product on the machine, and the product is processed by consuming parts in the part queue. After being processed, the product is unloaded by the operator and is moved to the warehouse.

For this manufacturing system, we can build three simulation models according to different modeling purposes. While both machining process and operator are modeled in detail in Model A, the operator's behaviors are abstracted in Model B, and in Model C the machining process for Machine 2 is omitted as well. These three models can be used for different modeling purposes, and observable variables from each model represent its modeling purposes: Using Model A, we can evaluate the operator's utilization as well as the performance of machines. Under the assumption that the operator's behaviors can be ignored, Model B is used for evaluating machining performance of the manufacturing system, and if we are interested in only the performance of Machine 1, Model C can be used.

Each model requires different modeling efforts. Since Model A is more complex than Model B or C, it takes more time and cost to model. However, Model A can be used for general purposes with simple abstractions instead of using either of the other two models. By setting the operator's loading and unloading times to zero and under the assumption that they use a same random number generator with the same initial seeds, the operator's

behavior in Model A can be nullified and Model A achieves logically the same results as Model B. In addition, by setting the processing time of Machine 2 to zero in Model A, it can also be used in place of Model C.

Therefore, Model A is important at the point of modeling productivity, since it covers the purposes of the other two models with abstractions. In other words, Model A is a general model representing the target system that can be used for various modeling purposes.

In later chapters, we will refer to the general simulation model as the one that has the high(est) fidelity that we can achieve. For a target system, because many models can be abstracted from the high fidelity model in which the results of system analysis are embedded, we can expect this approach to help improve modeling productivity.

However, in order to apply this approach to simulation modeling, we need theoretical foundations to define the underlying concepts and to validate the correctness of this approach. First, we need to define a metric for the fidelity of simulation models and study the nature of model fidelity. In addition, we also need to study abstraction techniques and their effects and evolve a theoretical framework to show whether the abstracted models from the high fidelity model are interchangeable with the models built from scratch.

Example II: Improving Productivity in the Dimension of Scope of System Domain

This example provides the conceptual ideas to develop automatic program generation through formal descriptions and domain knowledge for a specific domain.

Let a summation with step-increment be defined by

$$\sum_{\alpha}^{\gamma}{}^{\beta} = \sum_{i=0}^{k}(a+i\beta) = \alpha + (\alpha+\beta) + (\alpha+2\beta) + \cdots + (\alpha+k\beta)$$

where : α, β, γ are constant integers and $k = \lfloor (\gamma - \alpha)/\beta \rfloor$

From this definition, we can derive a property for efficient calculation: When β = 1, $\sum_{a}^{\gamma}{}^{1} = (a+\gamma)(\gamma - a + 1)/2$. Suppose a problem (**P1**) is given as follows:

$$\sum_{a}^{b}\sum_{c}^{e}{}^{d} \sum_{f}^{g}\sum_{h}^{i}\sum_{j}^{k}{}^{l}\sum_{m}^{n}{}^{o} = ?$$

Figure 1.2 P1: An Instance of Summation Problem

Since β ≠ 1, we cannot use the property for this problem; instead, we can build a computer model for solving this problem as follows:

```
S1: Computer Model for Solving Problem P1
          {    A = StepSum(j,l,k);
               B = StepSum(h,A,i);
               C = StepSum(f,B,g);

               D = StepSum(m,o,n);
               E = StepSum(c,d,e);
               F = StepSum(a,E,b);

               Answer = StepSum(F,C,D);
               Return Answer;
          }
```

Where, StepSum(α,β,γ) is a subroutine to calculate $\sum_{\alpha}^{\gamma}{}^{\beta}$

To build the computer model (**S1**), we should analyze problem (**P1**) to find the correct sequence of calculations, and develop a subroutine `StepSum`(α,β,γ) to calculate **S1** efficiently. However, if a new summation problem with a different structure arises, we can reuse only the subroutine `StepSum`(α,β,γ), but we have to analyze the structure of the new problem again to build a new computer model.

In this problem-solving framework, whenever a new problem is introduced, the modeler must analyze the problem again to find the correct sequence of calculation and build a corresponding computer model repeatedly. However, we can design a model generation procedure to create computer models automatically for any problems in this domain. The idea is explained in the following.

The summation problem domain (**PD**) is defined as a set of problems consisting of operator $\sum$ and its three parameters, in which the parameters can also consist of operator $\sum$ or integer parameters repeatedly. In the **PD**, the critical issue for automation is the identification of the precedence relationship in calculating the `StepSum`() function.

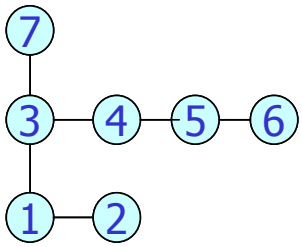


Figure 1.3 Tree T1 Representing Problem P1

A summation operation can start if all parameters of an operator ∑ are integer values. Therefore, any problem in this domain can be represented with a tree graph in which each node has no more than 3 branches. Each node represents a ∑ and each branch represents the evaluation of one of the operator's parameters. For example, problem P1 can be converted to a tree (**T1**) in Figure 1.3.

For a given tree structure, we can design an algorithm to find the sequence of calculation as follows:

Step1. Find a node linked with only one arc.
Step2. Delete the node
Step3. Repeat Step 1 and 2 until no nodes are linked with only one arc
Step4. Delete an arc linked with only one node
Step5. Repeat from Step 1 and 4 until no node is left

The sequence of deleting nodes is exactly matched with the proper sequence of calculation. This makes it possible to develop a model generation program, based on the tree structure to describe summation problems and the algorithm to find the sequence of calculation. The tree structure and parameters for describing a summation problem can be formally represented by an incidence matrix (**M**) and a data table (**T**). For **P1**, the corresponding formal descriptions are represented in Figure 1.4.

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	0	0	0	0	0
3	1	0	0	0	0	0	1
4	0	0	1	0	1	0	0
5	0	0	0	1	0	1	0
6	0	0	0	0	1	0	0
7	0	0	1	0	0	0	0

Incidence Matrix (M)

1	a	b	
2	c	e	d
3			
4	f	g	
5	h	i	
6	j	k	l
7	m	n	o

Data Table (T)

Figure 1.4 Formal Descriptions for P1: Incidence Matrix (M) and Data Table (T)

Therefore, since any problem in the domain can be represented by the incidence matrix and data table, we can build an automatic model generation procedure for any problem in the domain by using the algorithm to find sequence of calculation. This is due to the power of formal descriptions of problems using domain knowledge.

We can also use this approach on simulation modeling to improve the simulation modeling productivity. For a specific domain of manufacturing systems, we can find modeling components to capture domain knowledge, like the tree structure in the previous example. If the structure of modeling entities can be represented with formal structures (called a reference model for the domain), and if the principles and rules governing the behaviors of modeling entities can be designed as procedures (called a model generation procedure for the domain), we can build simulation models for any manufacturing system in a domain efficiently.

1.4 Research Objectives and Tasks

In this thesis, we focus on the two approaches for the systematic improvement of simulation modeling productivity. One approach is to adopt a wider scope of purpose for a target system to achieve various simulation results through abstraction. We call this *Virtual Factory Approach*, referring to a model with high(est) fidelity. The other approach is to adopt a wider scope of system domain. We call this *Reference Model Approach*. It is to build a simulation model efficiently for any system in a specific domain from the formal descriptions and domain knowledge.

In the Virtual Factory Approach, the challenge is to show its correctness. Although we can derive a simulation model easily from the high fidelity model with simple abstractions, we cannot be certain that the abstracted model behaves in a manner identical to the behavior of the simulation model built from scratch. Hence, the first research objective is as follows:

Research Objective 1: To develop a theoretical framework of the relationships between higher fidelity and lower fidelity models, and to provide justification that the models abstracted from a higher fidelity model are equivalent to various abstract simulation models for a target system.

The tasks defined to meet the stated objective are the following:

- *Develop a framework for simulation modeling.* This framework provides underlying foundation for interpreting the nature of simulation modeling by identifying components related to simulation modeling and relations among the components to represent simulation modeling process.
- *Define the relative fidelity indicator based on the simulation modeling framework.* The relative fidelity indicator is designed to provide a framework for comparing the fidelity of the models.
- *Characterize higher and lower fidelity models, and derive the formal validation for the Virtual Factory Approach.* Based on the simulation modeling framework and definition of relative fidelity of models, we can derive properties for the

relationships between higher and lower fidelity models to validate the Virtual Factory Approach.

In the Reference Model Approach, the challenge is to formalize the domain knowledge in the forms of a reference model and a model generation procedure. If we cannot formalize the common domain knowledge covering all systems in the domain, this approach cannot be deployed successfully. Therefore, the developer who designs a reference model and a model generation procedure should have domain knowledge as well as simulation modeling expertise, while model builders require only system knowledge.

As we pointed out in the introduction, the reason that manufacturing is one of the major areas of application for simulation modeling is because logical rules governing them are relatively better known than in other systems. Hence, in this thesis, we deploy the Reference Model Approach on manufacturing systems through formal domain knowledge. The second research objective is described as follows:

Research Objective 2: To develop the *Reference Model-based Simulation Modeling Methodology* for improving modeling productivity, and to deploy the methodology to the domain of *discrete part manufacturing systems*, which covers a large fraction of modern manufacturing systems.

The tasks defined to meet the stated objective are the following:

- *Analyze and compare simulation modeling methodologies.* This task highlights the necessities for the Reference Model Approach by comparing merits and demerits of current simulation methodologies.
- *Analyze the domain and design the reference model for discrete part manufacturing systems.* Through domain analysis, we find modeling entities that represent the behavior of systems sufficiently in the domain, and based on the results of domain analysis, we design a reference model to describe manufacturing systems in the domain formally.
- *Design an automatic model generation procedure*. We design a model generation procedure to generate a simulation model derived from the formal descriptions in the reference model and based on the domain knowledge derived from the domain analysis.

1.5 Organizations of the Thesis

The thesis consists of seven chapters. In Chapter II, we present a theoretical modeling framework for comparing the fidelity of simulation models, and define a relative fidelity indicator, which is used to derive some properties about simulation modeling and model fidelity. Based on the analysis of simulation modeling problem formulated under the simulation modeling framework, in Chapter III, we study the nature of model fidelity, explain the concept of virtual factory, and present model abstraction methods that are widely used in the Virtual Factory Approach. In Chapter IV we present the features of current simulation modeling methodologies, and analyze the merits and demerits of these

methodologies. Based on the results of the analysis, we propose a new modeling methodology called the Reference Model Approach by using a formal structure. The key concept of the Reference Model Approach is to build a proper reference model for a specific domain, which is a formal structure to represent an instance of the system in the domain, and a model generation procedure, which automatically creates a simulation model from formal descriptions based on common domain knowledge. In order to deploy this methodology to the domain of discrete part manufacturing systems, we develop a reference model for manufacturing (RMM) from the results of domain analysis. A detailed procedure for domain analysis and a reference model for manufacturing are presented in Chapter V. In Chapter VI, we develop a model generation procedure for RMM based on a simulation execution mechanism, which is well matched with the structure of RMM. Finally, in Chapter VII, we present the conclusions of this research and discuss future studies related to simulation methodology for improving simulation modeling productivity.

CHAPTER II

A THEORETICAL FRAMEWORK FOR COMPARING THE FIDELITY OF SIMULATION MODELS

2.1 Introduction

The basic idea of the *Virtual Factory Approach* is to build simulation models for a target system with various modeling purposes based on a general model with detailed or high fidelity. Hence, it is important to make sure the abstracted model, built from the general model, is equivalent to (or interchangeable with) the model built from scratch for specific purposes. Without the assurance that the behaviors of these models are equivalent, the Virtual Factory Approach cannot be used as a valid methodology to improve simulation modeling productivity. However, to show the behavioral equivalence of simulation models is extremely challenging.

Although the idea of equivalent simulation models may appear simple, it is difficult to formalize in an acceptable fashion since any practically useful definition of equivalence must be *testable*. That is, it should be possible to identify when two simulation models can be used interchangeably without actually having to run both models under all possible experimental conditions and compare their output results. Although there have been several definitions of behavioral equivalence (Overstreet 1982, Schruben 1983, Sargent 1988), none of them are testable. In 1992, Yücesan and Schruben presented an explicit and sensible definition of behavioral equivalence, and derived a property to

assess when it is safe to substitute one model for another by using a graph-theoretic specification of simulation models, called Simulation Graphs (Yücesan and Schruben 1992). However, not even their definition is testable in practice and all simulation models have to be built with Simulation Graphs for comparison. Therefore, since it is highly unlikely that any method can be constructed to determine whether simulation models are behaviorally equivalent or not, we need to design a surrogate method that is reasonable to determine the equivalence of simulation models.

Comparing the fidelity of simulation models can be an alternative for checking the equivalence of simulation models. Generally, fidelity means a metric for measuring the closeness to a real system. Since a model is often far removed from the faithfulness of a real system as it gets abstracted, fidelity becomes the opposite measurement of abstraction. Hence, if we can measure the fidelity of models without actual experiment, it can be an alternative to decide the equivalence of simulation models. So, it is important to define the fidelity of a model in a way that can be measured, and to develop a systematic way of comparing the fidelity of models for determining the behavioral equivalence of simulation models.

Although the term, *fidelity* has been widely used in simulation as an important attribute of a model, it cannot be measured in practice. There have been several studies on measuring fidelity of simulation models in absolute and quantitative ways, but there is still no consensus on a workable fidelity metric (Gross *et al*. 1999).

In this chapter, we propose a theoretical framework for comparing the fidelity of simulation models. In this framework, instead of measuring the fidelity of models in an absolute and quantitative way, we define the relative fidelity indicator of simulation models and provide a systematic way of comparing the fidelity of simulation models. The fidelity comparison framework in this research focuses on input and output interfaces and the variables of simulation models. It does not require any modeling formalism such as Simulation Graphs used in the previous research of Yücesan and Schruben (1992).

This chapter is organized as follows. In section 2.2, we discuss the literature for the fidelity of simulation models and model abstractions, and we explain a simulation modeling framework focusing on input and output observable variables in section 2.3. In section 2.4, we define the relative fidelity indicator, which is used for deriving properties related to the fidelity of simulation models in the next chapter.

2.2 Literature on Fidelity of Simulation Models and Model Abstractions

Fidelity has been an important attribute of models for evaluating the quality of simulation models (Pace 1998). However, compared to its importance in simulation, the nature of fidelity has not been researched well in literature. Recently, as interests in fidelity have increased, the *Fidelity Implementation Study Group* (Fidelity ISG) was organized by the *Simulation Interoperability Standard Organization* (SISO), and through a series of conferences, some of the research issues and results related to fidelity were identified. Quoting the definition of fidelity from the glossary of the Fidelity ISG, the fidelity of a simulation model is:

> **Fidelity**: 1) *The degree to which a simulation model reproduces the state and behavior of a real system in a measurable or perceivable manner*. 2) *A measure of the realism of a simulation model; faithfulness* (Gross 1999).

Gross *et al*. (1999) emphasized the justification for measuring the fidelity of simulation models from various viewpoints, and mentioned that measuring the fidelity of models would be potentially important in model selection in data rich domains such as manufacturing systems. As a preliminary study of fidelity, Schricker *et al*. (2001) proposed the Fidelity Evaluation Framework (FEF) to measure fidelity in an absolute and quantitative way, and Roza *et al*. (1998) presented a Fidelity Management Framework to provide a structural approach for specifying fidelity characterization, and compared previous research works that can be classified into two groups: singular fidelity metrics and complex multiple-dimensional fidelity metrics. After summarizing these research efforts, Fidelity ISG submitted the final report (Gross 1999) covering comprehensive areas for the fidelity of models. However, they could not propose systematic procedures to measure the fidelity of models as they intended.

Obstacles in Fidelity Measurement

Fundamentally, there are obstacles in measuring the fidelity of simulation models in an absolute and quantitative way. An obstacle comes from the incompleteness of measurable representation of a real system. In order to measure the fidelity of models, a target system should be represented measurably in all aspects. Without a comprehensive model that

provides a complete explanation of the behavior of a real system, the fidelity of a model cannot be measured in any absolute way. However, it is impossible to build such a model that includes all the factors of a real system. In literature, this hypothetical model is called a *base model* (Zeigler *et al*. 2000), and the practical model for surrogating the base model is called a *referent* (Gross *et al*.1999).

Although a referent includes as many factors as possible, it cannot include all the factors of a real system finally. So, there are simulation models having other factors that are not included in a referent. In this case, the measured fidelity of models based on the referent should be changed to a new referent model including new factors. After all, the fidelity metric based on a referent is a kind of conditional fidelity metric for simulation models that are comparable only under a given referent model. Especially, as the nature of a real system is articulated more and new knowledge of the system is added on, a referent model itself should be changed over time. Because of the increased number of factors, the measurable standard in a referent model can be changed. Due to the incompleteness of measurable representation of a real system, it may not be possible to measure the fidelity of models for a real system in an absolute and quantitative way.

Even though a real system is understood and represented completely in a measurable way, there is another obstacle in measuring the fidelity of simulation models. This is due to the multifaceted aspects of simulation models (Zeigler 1984). While a quantitative fidelity metric should be measured as a single number, factors describing models are multivariate. Hence, in order to measure the fidelity in an absolute and quantitative way, normalizing

weight factors are required. However, it is also difficult to decide the weight factors correctly, with which the sum of weighted value has the meaning of fidelity as a measure.

Simulation Model Abstraction

There have been several research activities addressing model abstraction that are closely related to the fidelity of models. In particular, Zeigler (1998) and Sevinc (1991) proposed a theory-based framework for understanding the issues in model abstraction, and Travers and Sevinc (1992) presented a number of approaches to extract information from simulation models of a system by abstracting its behavior. For abstraction techniques, Zeigler *et al*. (2000) identified four categories of simplification methods; Innis and Rexstad (1983) described 17 simplification techniques; and Frantz (1995) classified the comprehensive abstraction techniques with a taxonomic approach. For manufacturing systems, Brooks and Tobias (2000) presented several simplification techniques and showed that the simplification could be justified for specific average performance measures such as utilization in a case study. However, for other measures such as amount of work-in-process (WIP) and cycle time, introducing serious abstract errors is inevitable. One of the difficulties in valid abstraction is that it is hard to estimate the abstraction error prior to model execution, and the validation of model abstraction is dependent on the individual case.

2.3 Simulation Modeling Framework

Klir (1985) recognized four different levels of knowledge: i) *Source Level*, ii) *Data Level*, iii) *Generative Level*, and iv) *Structure Level*. Zeigler *et al*. (2000) employed a general

concept of system theory and identified useful ways to specify dynamic systems. These ways of system specification can be ordered in levels as in Table 2.1. Based on the system specification levels 3 and 4, they established a simulation modeling framework in which four basic entities (*real system*, *model*, *experimental frame*, and *simulator*) and their relationships are defined.

Table 2.1 System Specification Hierarchy (Zeigler *et al.* 2000)

Level	Specification Name	Corresponding to Klir's	What we know at this level
0	Observation frame	Source system	How to stimulate the system with inputs; what variables to measure and how to observe them over time
1	I/O behavior	Data system	Time-indexed data collected from a source system; consists of input and output sets
2	I/O function		Knowledge of initial state; given an initial state, every input stimulus produces a unique output
3	State transition	Generative system	How states are affected by inputs; given a state and an input what is the state after the input stimulus is over; what output event is generated by a state
4	Coupled components	Structure system	Components and how they are coupled together. The components can be specified at lower levels or can even be structure systems themselves – leading to hierarchical structure

However, this framework is too difficult to handle for comparing the model fidelity directly, because it deals with the complex state transition of simulation models in detail. To compare the fidelity of simulation models, we need to develop a simulation modeling framework with lower levels of system specifications. In this thesis, we focus on the input and output variables consisting of simulation models with data system level (specification level 1 or 2). The basic entities of this simulation modeling framework are

shown in Figure 2.1, which provides underlying foundations for comparing the fidelity of models and analyzing the properties of the fidelity metric.

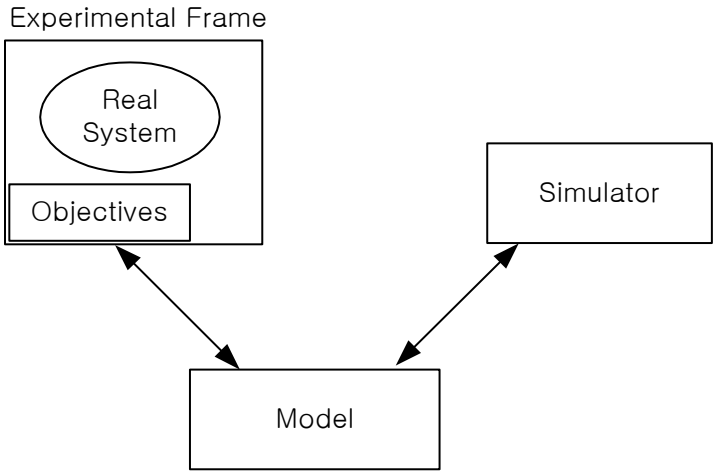


Figure 2.1 The basic entities in simulation modeling framework

2.3.1 Entities in Simulation Modeling Framework

Real System

A real system is the collection of objects that we have interest in modeling. It is viewed as a source of observable data in the form of variables. The values of each variable may change over time, and the pattern of changes is called the behavior of the variable. While the behaviors of some variables can be explained by a *logical process*, others cannot be explained logically. For example, part loading times on a machine are determined by a logical process with the state of the machine, availability of raw material, and the completion of prior job, etc. However, machine failure behaviors cannot be explained by

other observable variables logically. For such variables, we need to build models[1] to describe their behaviors inductively. As an example, machine failure can be modeled by a statistical distribution with parameters estimated from the data observations. Hence, the behavior of all observable variables in a system can be explained by logical processes or inductive models; both are called *logical rules* for the system in this framework.

Therefore, we can define a real system (**RS**) in the simulation modeling framework as a finite set of observable variables as follows;

$$\mathbf{RS} = \{o_1, o_2, o_3, \ldots, o_m\},$$

where o_i is an observable variable in a target system.

The observed data of variables in **RS** can be used as input data for executing a simulation model, or used as output data for comparing with the results of simulation. For a real system, **RS,** a set of observable variables and the known logical rules of the system are given as modeling foundation, on which all simulation models for the real system should be based.

Simulation Model

A simulation model can be viewed as a set of instructions to generate output behaviors from input data. From this definition, a simulation model can be divided into two parts: a *model structure* that is logic part implementing logical rules, and an *experimental frame* that is a set of observed data used for executing the model structure.

[1] This model is not a simulation model. Instead, it is a device to explain the behavior of the system.

Model Structure

In general, a model structure (or simply model) is a system specification, which can be implemented in various ways depending on modeling formalisms, which have different sets of modeling syntaxes and regulations for representing detailed state transition in model structures. Since every simulation package has its own modeling syntaxes and regulations, it can be regarded as a modeling formalism at level 3 and 4 in Zeigler's specification hierarchy. At these levels, a conceptual model can be implemented in many ways, and it is difficult to compare the simulation models.

Therefore, we would like to represent models at lower levels of the specification hierarchy that are independent of any modeling formalisms. At levels 1 and 2, we ignore the implementation-dependent state transition mechanisms and retain I/O behaviors and state variables. Although such a specification is not sufficient to implement simulation models, under the conditions that the logical rules in a real system are known explicitly and implemented correctly with given I/O behavior and state variables, this specification establishes a good foundation for simulation modeling and for the model comparison.

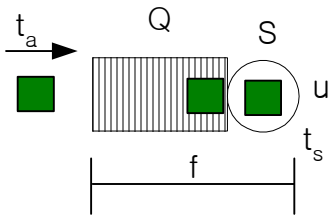
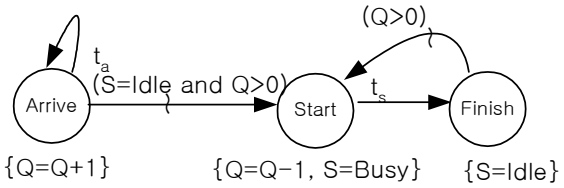


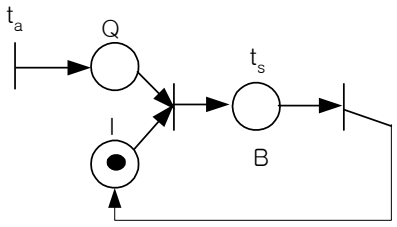
Figure 2.2 An example for a single queueing system

Consider the example shown in Figure 2.2, a single queueing system consisting of a queue and a server can be specified with i) state variables (**Q**: number of customers in

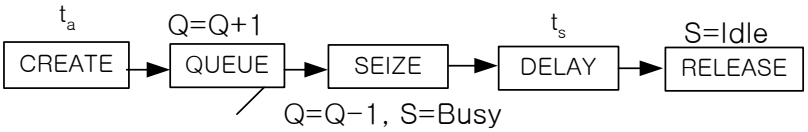
queue, and **S**: state of server), ii) input behavior variables ($\mathbf{t_a}$: inter-arrival time, $\mathbf{t_s}$: service time), and iii) output behavior variables (**f**: flow time, **u**: utilization of the server). Under the logical rules of the system, we can build simulation models with three different modeling formalisms shown in Figure 2.3. Although the state transitions are represented by a different formalism, given that logical rules are implemented correctly, the three models represent the system with same conceptual level through identifying I/O behavior and state variables.



(a) Event Graph Model for Single Queue System



(b) Petri-Net Model for Single Queue System



(c) Process-Oriented Model for Single Queue System (SIMAN Language)

Figure 2.3 Three different representations for single queue model

Since the state variables of each simulation model can be initialized with the same initial conditions, and the state variables can be regarded as output behavior variables, a model can be defined as two sets; input and output variables. All of these models in Figure 2.3

have same input variables ($\mathbf{t_a}$ and $\mathbf{t_s}$) and output variables (**Q**, **S**, **f**, and **u**) for representing the system with same conceptual level. To avoid confusion in naming, we use the term *interface* for the variables used in model specification. Hence, in our simulation modeling framework, a model, **M**, is denoted by:

$$\mathbf{M} = \langle \mathbf{IIF, OIF} \rangle,$$

where, **IIF** is the set of input interface variables, and **OIF** is the set of output interface variables.

Experimental Frame

An experimental frame consists of two sets. One is a set of observable variables that is used as input data for model structure. Another is a set of observable variables defined in **RS**, which corresponds to the output interface variables of model structure. For example, in the previous single queueing system, inter-arrival time ($\mathbf{t_a}$) and service time ($\mathbf{t_s}$) are input interface variables. To execute the model, the values corresponding to these input variables should be observable in the system. This is the set of input observable variables in an experimental frame. While executing a model, we can gather the output data from the model via output interface variables. For example, flow time (**f**) data can be gathered as the difference of times between entering and exiting the system. For these output interface variables of the model, there are corresponding observable variables in the real system. This is the set of output observable variables in the experimental frame.

In other words, an experimental frame (**EF**) plays a role to link a model (**M)** with a real system (**RS)** for simulation execution. **M** is a structure of input and output interfaces that

are not directly linked with **RS**. Hence, we need an experimental frame to have **M** executing under certain experimental conditions of **RS**. Within this frame, input observable variables are gathered from **RS**, and output results of **M** are compared with the observed values of output observable variables in **RS**.

An experimental frame (**EF**) is characterized as sets of input and output variables of interests that are observable from a real system (**RS**), and denoted as follows:

$$\mathbf{EF} = < \mathbf{IV}, \mathbf{OV} >,$$

where **IV** is the set of input variables observable from **RS**, i.e. $\mathbf{IV} \subseteq \mathbf{RS}$, and **OV** is the set of output variables observable from **RS**, i.e. $\mathbf{OV} \subseteq \mathbf{RS}$, and $\mathbf{IV} \cap \mathbf{OV} = \varnothing$.

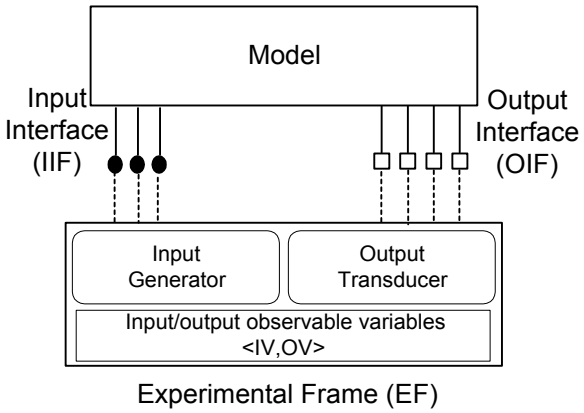


Figure 2.4 The illustration of model and experimental frame

For linking with models, an experimental frame needs two transform functions. One is called *generator*, the role of which is to transform observed data from **IV** defined in **EF** to modeled input data that are transferred through **IIF** of **M**. Another function to be defined in the experimental frame is *transducer*, and its role is to transform observed output results generated from **OIF** of **M** to the output results in the form of **OV** in **EF**.

Hence, for a given **EF**, we can get two contents of **OV**: One from executing **M** under **EF**, and another one from **RS**. These two sets of output data are compared and analyzed for model validation. For convenience of notation, we denote these two functions as follows:

Generator: **EF.IV→M.IIF**, and *Transducer*: **M.OIF→EF.OV**

Figure 2.4 illustrates the relations between a model and an experimental frame with two transform functions.

Simulator

A simulator is a computerized program executing a simulation model under certain experimental conditions specified by an **EF**. In a simulator, **EF** generates input sequences for executing **M**, and it also observes and stores the output behaviors of **M** through interfaces. Hence, in this framework, a simulator represents the simulation results under certain experimental conditions, which is denoted by **M**||**EF**.

Objectives

Objectives of simulation study can be represented by a subset of output variables in an experimental frame. Since objectives measures the output behaviors of a simulation model, the objectives can be achieved by observing output variables of the experimental frame, which are comparable to observable variables in a real system. In this framework, the objectives include not only the set of observable variables, but also any function of output observable variables implicitly. Hence, the objective of a simulation study is denoted as follows:

$$\mathbf{Obj} = \{obj_1, obj_2, \ldots, obj_n\},$$

where obj_k is an observable variable in **RS**.

2.3.2 Relations between Entities in Simulation Modeling Framework

Relationship between Model (**M**) and Experimental Frame (**EF**)

A model (**M**) is represented as a structure of input and output interfaces, and an experimental frame (**EF**) is represented by two sets of input and output observable variables in a real system (**RS**). Between models and experimental frames, there are *applicability* relations, if the conditions for experimentation required by a frame can be enforced in a model. In other words, **M** should have sufficient input interfaces to accept semantically matching input data from the *generator* in **EF**, and **M** should fill all output variables in **EF** by the *transducer* to convert output results flowing through output interfaces of **M**. If any input data generated from **EF** are not used in **M**, it means that the experimental conditions defined in **EF** are not reflected completely in **M**. On the other hand, if we cannot get some output results from **M** that should be compared with observable variables defined in **EF**, **M** is not valid for **EF**. Hence, formally, the applicability relation is defined as follows:

For any given **EF** and **M**, if *Generator*(**EF.IV**) $\subseteq$ **M.IIF** and *Transducer*$^{-1}$(**EF.OV**) $\subseteq$ **M.OIF** and these interfaces are semantically matched, then there is applicability relation between **EF** and **M**, and it is denoted as follows:

$$\mathbf{EF} \xrightarrow{a} \mathbf{M}$$

It is said that experimental frame, **EF** is *applicable* to **M**.

Between models and experimental frames, there can be many to many relationships with respect to applicable relations. For an experimental frame, there may be multiple models

for which the experimental frame is applicable, and a model can be executed under various experimental frames.

Relationships among Experimental Frames (**EF**'s)

An experimental frame is specified by sets of input and output observable variables, and it represents experimental conditions in which we are interested. The degree to which one experimental frame is more restrictive in its conditions than another is formulated in the *derivability* relation. A more restrictive frame leaves less room for experimentation or observation than one from which it is derivable. In other words, if an experimental frame has sufficient information to derive another experimental frame, then there is derivability relationship between these two frames. Formally, the derivability relation is defined as follows:

For a given $\mathbf{EF_a} = \langle \mathbf{IV_a}, \mathbf{OV_a} \rangle$, $\mathbf{EF_d} = \langle \mathbf{IV_d}, \mathbf{OV_d} \rangle$, if there exist functions to transform from the contents of $\mathbf{IV_d}$ and $\mathbf{OV_d}$ to those of all input observable variables in $\mathbf{IV_a}$, and from the contents of $\mathbf{OV_d}$ to those of all output observable variables in $\mathbf{OV_a}$, then it is said that $\mathbf{EF_a}$ is *derivable* from $\mathbf{EF_d}$, and the relation is denoted by:

$$\mathbf{EF_d} \xrightarrow{d} \mathbf{EF_a}$$

Full Frame for a model **M**

Full frame ($\mathbf{EF^*_M}$) of model **M** is an experimental frame of the maximal experimental conditions on which **M** can be experimented. Full frame is a characteristic attribute of a model, and it is defined formally as follows:

$$\mathbf{EF}^*_{\mathbf{M}} = \{\mathbf{EF} \mid \mathbf{EF} \xrightarrow{d} \mathbf{EF_j}, \text{ for } \forall\ \mathbf{EF_j} \xrightarrow{a} \mathbf{M}, \text{ and } \mathbf{IV_j} \cup \mathbf{OV_j} \in \mathbf{RS}\}$$

[Proposition 2.1] There is no experimental frame, $\mathbf{EF}^+ \neq \mathbf{EF}^*_{\mathbf{M}}$, in **RS** such that $\mathbf{EF}^+ \xrightarrow{d} \mathbf{EF}^*_{\mathbf{M}}$, and $\mathbf{EF}^+ \xrightarrow{a} \mathbf{M}$.

Proof: Obvious from the definition.

This proposition implies that the full frame of a model restricts the valid experimental frame for the model in **RS**. It means that any valid frame applicable to a model should be derivable from the full frame of the model, and if there is any frame that is not derivable from the full frame, it uses invalid variables that are not observable from the given **RS**.

Total Set of Models for a Real System **RS**

In the simulation modeling framework, since models are characterized by I/O interface variables, for a given real system (**RS**), there are a finite number of models of which full frames are applicable to the **RS**. This set is called the total set of models ($\mathbf{M_{RS}}$) for **RS**, and is defined as follows:

$$\mathbf{M_{RS}} = \{\mathbf{M} \mid \mathbf{EF}^*_{\mathbf{M}} \xrightarrow{a} \mathbf{RS}\}$$

Set of Models for an Experimental Frame **EF**

For a given experimental frame (**EF**), there may be a number of models to which the experimental frame is applicable. The set of such models is called a set of models for an experimental frame, and it is defined as follows:

$$\mathbf{M_{EF}} = \{\mathbf{M} \mid \mathbf{EF} \xrightarrow{a} \mathbf{M}, \text{ for } \forall\ \mathbf{M} \in \mathbf{M_{RS}}\}$$

Set of Experimental Frames for a Model **M**

For a given model (**M**), there may be a number of experimental frames that are applicable to **M**. This set of experimental frames is called a set of experimental frames for a model, and it is defined as follows:

$$\mathbf{EF_M} = \{\mathbf{EF} \mid \mathbf{EF} \xrightarrow{a} \mathbf{M}, \text{ for } \forall\ \mathbf{EF} \text{ s.t. } \mathbf{EF.IV} \text{ and } \mathbf{EF.OV} \subset \mathbf{RS}\}$$

Relationship between Objective (**Obj**) and Experimental Frame (**EF**)

If there is an **EF** from which a set of objectives (denoted by **Obj)** is derivable, then it is said that **Obj** is *achievable* by **EF**, denoted by $\mathbf{EF} \xrightarrow{d} \mathbf{Obj}$. Actually, a set of objectives is a kind of experimental frame consisting of only a set of output observable variables. Hence, we can use the derivability relation without modifications for achievability relationship.

2.4 Comparing Fidelity of Models

In this section, we deal with the comparisons of fidelities of simulation models. However, since it is impractical to measure fidelity in absolute and quantitative ways, here we focus on comparing the relative relationship of the fidelity between models. We call this *relative fidelity indicator*, which will be defined in the formal framework described in the previous sections. The relative fidelity indicator provides a systematic procedure for comparing the fidelity of models. We illustrate the conceptual meaning of model comparison with the following example.

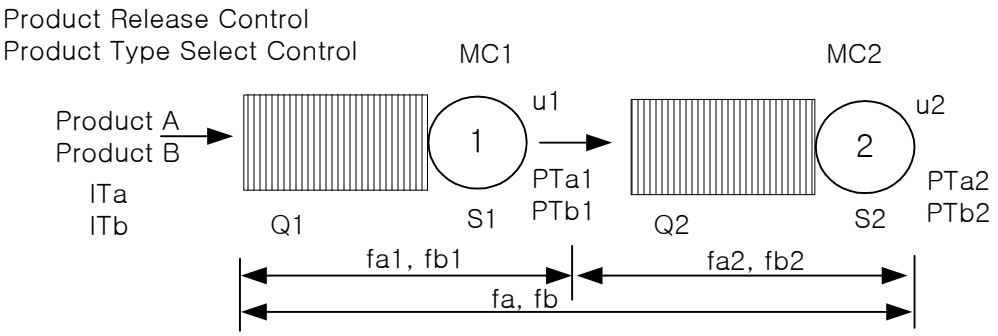


Figure 2.5 Two-machine serial line production system, and observable variables

Table 2.2 Observable variables of two-machine serial line production system

Observable Variables	Descriptions
IT_a	Inter releasing time for Product A
IT_b	Inter releasing time for Product B
PT_{a1}	Product A Processing time on MC_1
PT_{b1}	Product B Processing time on MC_1
PT_{a2}	Product A Processing time on MC_2
PT_{b2}	Product B Processing time on MC_2
Q_1	Number of products in MC1 queue
Q_2	Number of products in MC2 queue
S_1	State of MC_1 (0, if MC_1 is idle, 1, otherwise)
S_2	State of MC_2 (0, if MC_2 is idle, 1, otherwise)
f_{a1}	Cycle time for Product A in MC_1
f_{b1}	Cycle time for Product B in MC_1
f_{a2}	Cycle time for Product A in MC_2
f_{b2}	Cycle time for Product B in MC_2
f_a	Flow time for Product A
f_b	Flow time for Product B
u_1	Utilization of MC_1
u_2	Utilization of MC_2

Example: Two-Machine Flow Line System

System Descriptions

As Figure 2.5 shows, the example manufacturing system (**RS**) consists of two machines. Two types of product denoted as Product A and Product B are produced. Products are

released according to two logical rules (control logics) that deal with when and which product type should be released, and are processed sequentially on Machine 1 (MC_1) and Machine 2 (MC_2). Two control logics, *Product Release Control* and *Product Type Select Control* are known, and all variables defined in Table 2.1 are observable in the system operations. It is assumed that other variables are not observable from the system. Hence, according to the simulation modeling framework, a manufacturing system, **RS** is represented as the set of all observable variables in Table 2.2 with two control logics.

Modeling and Abstraction

Based on the data from observable variables and logical rules, we can build several models for simulating the target system. Among these models, the most detailed model is to implement all logical rules governing the system operations as they are without any structural and behavioral abstractions. In this example, we can build the most detailed model by correctly implementing two control logics, product release control and product type select control, to generate all output behaviors that are observable in the manufacturing system. For this model, since the control logics are implemented explicitly inside of the model, the relevant input data to execute the model are only processing times, which can be obtained by observing variables such as PT_{a1}, PT_{a2}, PT_{b1}, and PT_{b2} in **RS**. Let the model be denoted by $\mathbf{M_A}$. Then, its corresponding full frame $\mathbf{EF^*_{MA}}$ is described as follows.

$$\mathbf{EF^*_{MA}} = < \mathbf{IV_A} = \{PT_{a1}, PT_{b1}, PT_{a2}, PT_{b2}\},$$
$$\mathbf{OV_A} = \{IT_a, IT_b, Q_1, Q_2, S_1, S_2, f_{a1}, f_{b1}, f_{a2}, f_{b2}, u_1, u_2\}\}$$

By using various abstraction methods (Frantz 1995, Innis and Rexstad 1983, Zeigler 1998), we can build many abstracted models representing the same system. In this example, we apply two abstraction methods: i) ignoring product types and ii) randomized control[2] for product release.

i) Ignoring product types:

If we ignore product types, we do not need to implement product type select control explicitly in modeling. However, since the product types are not identified in the model, processing time data for each machine should be merged without classifying the product types. Hence, instead of implementing the product type select control, we can easily build an abstracted model using merged processing time data. Since PT_{ij} variables are observable, PT_i variables for merged processing times are also observable in **RS**.

ii) Randomized control for product release:

Product release control is to decide times to release a product to Machine 1. Instead of implementing the control logic explicitly, we can imitate the control behavior with randomized control. By observing product inter-arrival times, we can fit a statistical distribution and parameters imitating control behaviors. The implementation of randomized control is much easier than the implementation of the control logic explicitly. In this abstraction, product inter-arrival time variables (IT_a, IT_b) are used for fitting a statistical distribution and estimating parameters in the randomized control abstraction.

[2] The *randomized control* means a method to abstract a logical rule in modeling with replacing the implementation of the logical rule by generating pseudo random behaviors that follow a statistical distribution with parameters, which are estimated from the observation of behavior of the logical rule.

Hence, these variables are not used as output variable any longer, but they are used as input variables in this experimental frame.

By applying two abstraction methods, we can build four different models illustrated in Figure 2.6. Model A ($\mathbf{M_A}$)is the model in which all control logics are implemented explicitly. *Product Type Select Control* and *Product Release Control* are abstracted in Model B ($\mathbf{M_B}$) and Model C ($\mathbf{M_C}$), respectively, and both control logics are abstracted in Model D ($\mathbf{M_D}$). These four models are used for comparing the fidelity of models, and full frames ($\mathbf{EF^*}$s) for these models are listed in Table 2.3.

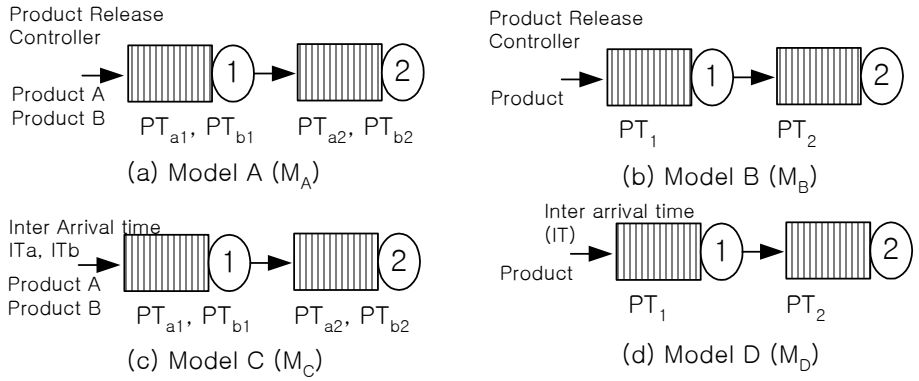


Figure 2.6 Four models for two-machine serial line production system

Table 2.3 Descriptions of four models and their full frames

Model (M)	Description	$\mathbf{EF^*_M.IV}$	$\mathbf{EF^*_M.OV}$
$\mathbf{M_A}$	Two control logics are implemented explicitly without any abstraction	PT_{a1}, PT_{b1}, PT_{a2}, PT_{b2}	IT_a, IT_b, Q_1, Q_2, S_1, S_2, f_{a1}, f_{b1}, f_{a2}, f_{b2}, u_1, u_2
$\mathbf{M_B}$	Product type is ignored in modeling, so it uses merged processing time for each machine	PT_1, PT_2	IT_a, IT_b, Q_1, Q_2, S_1, S_2, f_1, f_2, u_1, u_2
$\mathbf{M_C}$	Product Release Control is abstracted with randomized control using product inter arrival times.	IT_a, IT_b, PT_{a1}, PT_{b1}, PT_{a2}, PT_{b2}	Q_1, Q_2, S_1, S_2, f_{a1}, f_{b1}, f_{a2}, f_{b2}, u_1, u_2
$\mathbf{M_D}$	Both control logics are abstracted	IT_a, IT_b, PT_1, PT_2	Q_1, Q_2, S_1, S_2, f_1, f_2, u_1, u_2

Comparison of Models

Conceptually, fidelity indicates how much a model is not abstracted from a real system. If a model is more abstracted, the model has less fidelity. From this point of view, we can say that Model A has higher fidelity than any of the other three models, and Model B and Model C have higher fidelity than Model D, because more abstraction methods are applied to Model D. However, for Model B and Model C, it is hard to decide which one has higher fidelity according to this standard.

How can we make a systematic procedure to compare the fidelity of models, which correspond to the conceptual comparison? The derivability relation between full frames of the models being compared can answer this question. For example, consider two models, Model C and Model D, for comparing model fidelity. The full experimental frames for Model C and D are:

$$\mathbf{EF}^{*}_{\mathbf{MC}} = \langle \mathbf{IV_C} = \{IT_a, IT_b, PT_{a1}, PT_{b1}, PT_{a2}, PT_{b2}\},$$
$$\mathbf{OV_C} = \{Q_1, Q_2, S_1, S_2, f_{a1}, f_{b1}, f_{a2}, f_{b2}, u_1, u_2\}\rangle, \text{ and}$$
$$\mathbf{EF}^{*}_{\mathbf{MD}} = \langle \mathbf{IV_D} = \{IT_a, IT_b, PT_1, PT_2\},$$
$$\mathbf{OV_D} = \{Q_1, Q_2, S_1, S_2, f_1, f_2, u_1, u_2\}\rangle.$$

While the contents of input observable variables, PT_1 and PT_2, can be derivable from the contents of PT_{a1}, PT_{b1}, and PT_{a2}, PT_{b2}, the opposite direction is not possible. As with like input variables, the contents of output observable variables, f_1 and f_2, can be derived from the contents of f_{a1}, f_{b1}, and f_{a2}, f_{b2}. Conceptually, it means that Model C requires more

input variables for modeling and generates more output contents than Model D. Since Model C uses more inputs and generates more outputs that cover the outputs of Model D, we can conclude that Model C has higher fidelity than Model D.

However, there is another point of consideration for the derivability relation in comparing the fidelity of Model A and Model C. The full experimental frames for Model A and C are as follows:

$$\mathbf{EF}^{*}_{\mathbf{MA}} = <\mathbf{IV_A}=\{PT_{a1}, PT_{b1}, PT_{a2}, PT_{b2}\},$$
$$\mathbf{OV_A}=\{IT_a, IT_b, Q_1, Q_2, S_1, S_2, f_{a1}, f_{b1}, f_{a2}, f_{b2}, u_1, u_2\}>, \text{ and}$$
$$\mathbf{EF}^{*}_{\mathbf{MC}} = <\mathbf{IV_C}=\{IT_a, IT_b, PT_{a1}, PT_{b1}, PT_{a2}, PT_{b2}\},$$
$$\mathbf{OV_C}=\{Q_1, Q_2, S_1, S_2, f_{a1}, f_{b1}, f_{a2}, f_{b2}, u_1, u_2\}>.$$

In this case, the input variables for Model A are PT_{a1}, PT_{b1}, PT_{a2}, and PT_{b2} as for Model C. However, while the product release control logic is implemented explicitly in Model A, the control logic is substituted by randomized control with estimated distributions from product inter arrival time variables (IT_a and IT_b) in Model C. It means that these variables that are output observable variables in Model A are used for abstracting *Product Release Control* in Model C as input observable variables. Hence, in comparison of model fidelity, we should consider output variables if they were used as input variables of another model. In this example, the contents of input variables ($\mathbf{IV_C}$) in Model C can be derived from the input and output variables ($\mathbf{IV_A} \cup \mathbf{OV_A}$) in Model A, and the contents of output variables ($\mathbf{OV_C}$) in Model C can be derivable from the contents of output variables ($\mathbf{OV_A}$) in

Model A. However, the opposite derivation is not possible. Hence, we can conclude that Model A has higher fidelity than Model C with the derivability relations, which corresponds to the result of conceptual comparison.

In the case of comparing Models B and C, it is difficult to decide which one has higher fidelity even conceptually, because while in some aspects, Model B has higher fidelity than Model C, in some other aspects, Model B has lower fidelity than Model C. Comparing model fidelity with derivability relation also reflects this phenomenon. Since there is no derivability relation between Model B and Model C, we cannot decide which one has higher fidelity with derivability relation just like the result of conceptual comparison.

2.5 Relative Fidelity Indicator

The last the example illustrated the relations between a lower fidelity model and a higher fidelity model. The key elements to influence the fidelity of models are observable variables and logical rules in a real system. A logical rule is a known fact of how to control the behaviors of a real system. To implement a logical rule explicitly in a model, the required data for the logical rule should be observable in the system. If all of the logical rules are implemented explicitly in a model, then the model has the highest fidelity under given conditions.

However, because of the complexity or economical reasons of model development, some logical rules may be abstracted, and so the fidelity of the model is lessening. The ways of

abstracting logical rules should be justified and based on the data that we can observe from a real system. In other words, if a logical rule is abstracted, the input data for the abstracted logical rule should be observable in the real system, and its output data also should be comparable to the observable results of the real system. An abstracted model does not provide more I/O data than the original model before abstraction. Hence, based on this rationale, derivability relation between experimental frames is a systematic method of checking model fidelity (or abstraction).

Although this method cannot evaluate the fidelity of a model absolutely and quantitatively, it can provide information as to which model has higher or lower fidelity, or even whether the fidelity of models is *comparable* or not. We call this property *relative fidelity*, and it is defined in the simulation modeling framework as follows:

[Definition 2.1] Relative Fidelity Indicator

For models $\mathbf{M_1}$, $\mathbf{M_2}$, and experimental frames $\mathbf{EF_1}$ and $\mathbf{EF_2}$ defined on a **RS**, if $\mathbf{EF_1} \xrightarrow{a} \mathbf{M_1}$, $\mathbf{EF_2} \xrightarrow{a} \mathbf{M_2}$, and $\mathbf{EF_1} \xrightarrow{d} \mathbf{EF_2}$, then it is said that $\mathbf{M_1||EF_1}$ has higher fidelity than $\mathbf{M_2||EF_2}$ and denoted by:

$$\text{Fidelity}(\mathbf{M_1||EF_1}) \geq \text{Fidelity}(\mathbf{M_2||EF_2})$$

In the special case that $\mathbf{EF_1} \equiv \mathbf{EF_2} \equiv \mathbf{EF}$, then Fidelity($\mathbf{M_1||EF}$) = Fidelity($\mathbf{M_2||EF}$).

Figure 2.7 illustrates fidelity relations between simulation models graphically.

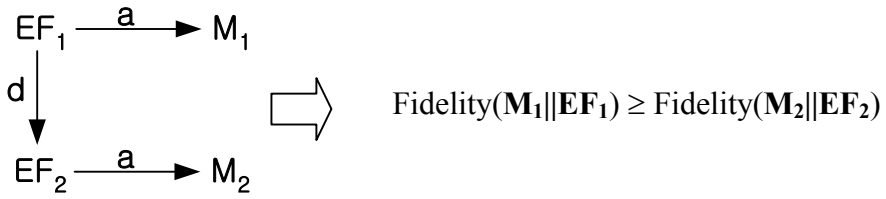


Figure 2.7 Graphical illustration for relative fidelity between $M_1||EF_1$ and $M_2||EF_2$

From the definition of the relative fidelity, the following properties are derived naturally.

[Proposition 2.2] Equal Fidelity

For models $\mathbf{M_1}$, $\mathbf{M_2}$, and experimental frames $\mathbf{EF_1}$, and $\mathbf{EF_2}$ defined on a **RS** with Fidelity($\mathbf{M_1}||\mathbf{EF_1}$) ≥ Fidelity($\mathbf{M_2}||\mathbf{EF_2}$). If $\mathbf{EF_2} \xrightarrow{a} \mathbf{M_1}$, then Fidelity($\mathbf{M_1}||\mathbf{EF_2}$) = Fidelity($\mathbf{M_2}||\mathbf{EF_2}$).

Proof: Obvious from the definition.

[Proposition 2.3] Model Fidelity

For the full frames for $\mathbf{M_1}$, $\mathbf{M_2} \in \mathbf{M_{RS}}$, if $\mathbf{EF^*_{M1}} \xrightarrow{d} \mathbf{EF^*_{M2}}$ or $\mathbf{EF^*_{M2}} \xrightarrow{a} \mathbf{M_1}$, then Fidelity($\mathbf{M_1}$) ≥ Fidelity($\mathbf{M_2}$).

Proof: Obvious from the definition.

2.6 Summary

In this chapter, we developed a formal simulation modeling framework based on I/O observable variables, which is useful to understand basic issues and problems encountered while performing simulation modeling. Since it is practically impossible to check whether two models are interchangeable or not, we introduce the concept of

fidelity as a surrogate. However, it is also difficult to measure the fidelity of models absolutely and quantitatively. Hence, within the simulation modeling framework, we defined relative fidelity based on derivability relations between I/O observable variables defined in experimental frames. In the next chapter, we derive some of the properties of simulation modeling and fidelity by using the relative fidelity indicator within the proposed simulation modeling framework.

CHAPTER III

ANALYSIS OF SIMULATION MODELING PROBLEM AND THE CONCEPT OF VIRTUAL FACTORY

In this chapter, we derive some of the properties related to simulation modeling and the fidelity of simulation models. Based on the relative fidelity indicator and theoretical modeling framework developed in the previous chapter, we formulate a simulation modeling problem to find the best model and experimental frame to achieve given modeling objectives under a reasonable assumption. By analyzing the simulation modeling problem, we derive properties related to simulation modeling and fidelity of simulation models, and based on these properties, we explain the virtual factory concept in light of improving simulation modeling productivity.

3.1 Simulation Modeling Problem

For a given set of objectives, **Obj**, the goal of simulation modeling is to find a model **M,** and an experimental frame **EF**, from alternatives to achieve the **Obj**. The total set of alternative models and experimental frames in a **RS** is denoted by:

$$\{(\mathbf{m}, \mathbf{ef}) \mid \mathbf{ef} \xrightarrow{a} \mathbf{m}, \text{ for } \forall\ \mathbf{ef} \xrightarrow{d} \mathbf{Obj}, \mathbf{m} \in \mathbf{M_{RS}}\}.$$

Among these alternative models and experimental frames, in order to find the best model ($\mathbf{M}^*$) and experimental frame ($\mathbf{EF}^*$), we need to evaluate models under experimental frames by which the given modeling objectives can be achieved. In addition, there are

other practical constraints such as limitations of modeling, experiment, and execution times. Hence, together with these constraints, for a given set of objectives **Obj**, a simulation modeling problem can be formulated as follows:

Simulation Modeling Problem (SMP)

For a given set of objectives, **Obj** and modeling constraint constants: *ModelingTimeLimit*, *ExperimentTimeLimit*, and *ExecutionTimeLimit*;

$$\underset{M,EF}{Min} \quad \text{Eval}(\mathbf{M}||\mathbf{EF})$$

s.t.

$$\mathbf{M} \in \mathbf{M_{Obj}} = \{\mathbf{m} \mid \mathbf{ef} \xrightarrow{a} \mathbf{m}, \text{ for } \forall\ \mathbf{ef} \xrightarrow{d} \mathbf{Obj}, \mathbf{m} \in \mathbf{M_{RS}}\}$$

$$\mathbf{EF} \in \mathbf{EF_M}$$

$$\text{ModelingTime}(\mathbf{M}) \le \textit{ModelingTimeLimit}$$

$$\text{ExperimentTime}(\mathbf{EF}) \le \textit{ExperimentTimeLimit}$$

$$\text{ExecutionTime}(\mathbf{M},\mathbf{EF}) \le \textit{ExecutionTimeLimit}$$

3.1.1 Difficulties in SMP

One of the difficulties in simulation modeling problem is that evaluation functions cannot be computed without actual execution. For example, Eval(**M**||**EF**) is a function to evaluate the performance of model **M** under a frame **EF**. Generally, Eval is to compare the differences of **M**||**EF** and **RS**||**EF**. However, it is difficult to quantify the measurement. Even if the measurement is quantifiable, it is achievable only after executing many alternative models. Further, the evaluation functions (i.e. ModelingTime,

ExperimentTime, and ExecutionTime) are highly dependent on the measuring conditions such as modeling tools and simulation execution platforms.

In addition, since we can not identify the behavioral pattern of Eval(**M**||**EF**) with respect to **M** and **EF** without actual execution, enumerating all alternatives by search methods is the only way to find and guarantee the best model and experimental frame. In addition, the fact that it takes a long time to build these alternative models is another difficulty in finding $\mathbf{M}^*$ and $\mathbf{EF}^*$ of SMP. Hence, it is not practically possible to find the best model and experimental frame by searching and evaluating alternative models (Brooks and Tobias 1996).

3.1.2 Assumption for SMP

Evaluating a model is influenced not only by the attributes of the model but also the quality of input data, random number generator, and other factors in simulation execution platform. The relative fidelity indicator defined in the previous chapter is independent of those factors that are difficult to acquire or control, and it can be a good attribute to establish the relationship between models and their evaluations. Since i) relative fidelity indicator is based on the inclusive relation that input and output data of the lower fidelity model are derivable from those of higher fidelity model and ii) it is intuitive that the simulation results of higher fidelity model are closer to the output behaviors of a real system than a lower fidelity model, we can state a fidelity assumption in symbolic form as follows:

[Assumption 3.1] Fidelity Assumption

If Fidelity($\mathbf{M_d}||\mathbf{EF_d}$) ≥ Fidelity($\mathbf{M_a}||\mathbf{EF_a}$), i.e. if $\mathbf{EF_d} \xrightarrow{a} \mathbf{M_d}$, $\mathbf{EF_a} \xrightarrow{a} \mathbf{M_a}$, and $\mathbf{EF_d} \xrightarrow{d} \mathbf{EF_a}$, then the results of simulation execution, $\mathbf{M_d}||\mathbf{EF_d}$ is closer to results of real system than $\mathbf{M_a}||\mathbf{EF_a}$. In the case of $\mathbf{EF_d}=\mathbf{EF_a}=\mathbf{EF}$, then simulation results of $\mathbf{M_d}||\mathbf{EF}$ is equivalent to those of $\mathbf{M_a}||\mathbf{EF}$, and it is denoted by $\mathbf{M_d}||\mathbf{EF} \equiv \mathbf{M_a}||\mathbf{EF}$.

This axiom means that more abstractions or simplifications generate less realistic results. If an **EF** is applicable to model **M**, it means that all input data generated by the **EF** are accepted by **M**, and all output behavior data specified in the **EF** can be observed from **M**. In addition, if the input data and output data are mapped with the same semantics to the two different models to which the same experimental frame is applicable, then we consider that the results of simulation for both models are logically same.

3.1.3 Analysis of SMP

In SMP, the main decision is to find the best model, $\mathbf{M}^*$ and the best experimental frame, $\mathbf{EF}^*$ that satisfies given **Obj**. Under the fidelity assumption, we can observe an optimality relationship between $\mathbf{M}^*$ and $\mathbf{EF}^*$. In this analysis, we ignore the constraints about modeling, experiment and execution times.

[Proposition 3.1] Optimum Experimental Frame

For a given single set of objectives, **Obj**, if a model $\mathbf{M}^*$ is an optimal model in a SMP, and its corresponding full frame $\mathbf{EF}^*_{\mathbf{M}^*}$ is achievable to **Obj** and applicable to **RS**, i.e.

($\{\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{d} \mathbf{Obj}\} \wedge \{\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{a} \mathbf{RS}\}$), then $\mathbf{EF}^*_{\mathbf{M}*}$ is an optimal experimental frame for the SMP with $\mathbf{M}^*$.

Proof: (by counter evidence)

Suppose that $\mathbf{EF}^*_{\mathbf{M}*}$ is not an optimal experimental frame for model $\mathbf{M}^*$. Then, there exists an optimal frame $\mathbf{EF}^* \neq \mathbf{EF}^*_{\mathbf{M}*}$, which is applicable to $\mathbf{M}^*$. There are three possible cases: i) $\mathbf{EF}^* \xrightarrow{d} \mathbf{EF}^*_{\mathbf{M}*}$, ii) $\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{d} \mathbf{EF}^*$, or iii) No derivable relation between $\mathbf{EF}^*$ and $\mathbf{EF}^*_{\mathbf{M}}$

For i), due to the definition of full frame, $\mathbf{EF}^*$ cannot be applicable to $\mathbf{M}^*$. This is a contradiction to the premise that $\mathbf{EF}^* \xrightarrow{a} \mathbf{M}^*$.

For ii), $\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{d} \mathbf{EF}^*$ means that there is a mapping from $\mathbf{EF}^*_{\mathbf{M}*} = <\mathbf{IV}^*_{\mathbf{M}*}, \mathbf{OV}^*_{\mathbf{M}*}>$ to $\mathbf{EF}^* = <\mathbf{IV}^*, \mathbf{OV}^*>$. Unless $\mathbf{EF}^*_{\mathbf{M}}$ is equivalent to $\mathbf{EF}^*$, the Fidelity($\mathbf{M}^*||\mathbf{EF}^*_{\mathbf{M}*}$) > Fidelity($\mathbf{M}^*||\mathbf{EF}^*$). Hence, the evaluation of $\mathbf{M}^*||\mathbf{EF}^*$ cannot be better than $\mathbf{M}^*||\mathbf{EF}^*_{\mathbf{M}*}$ by Assumption 3.1.

For iii), let $\mathbf{EF}^{**}$ be a superimposing experimental frame defined by $<\mathbf{IV}^* \cup \mathbf{IV}^*_{\mathbf{M}*}, \mathbf{OV}^* \cup \mathbf{OV}^*_{\mathbf{M}*}>$ and $\mathbf{M}^{**}$ be its corresponding model, then $\mathbf{EF}^{**} \xrightarrow{d} \mathbf{EF}^*$ and $\mathbf{EF}^{**} \xrightarrow{d} \mathbf{EF}^*_{\mathbf{M}*}$. Since $\mathbf{EF}^{**} \xrightarrow{a} \mathbf{M}^{**}$and $\mathbf{EF}^* \xrightarrow{a} \mathbf{M}^*$, the evaluation of $\mathbf{M}^{**}||\mathbf{EF}^{**}$ is better than both $\mathbf{M}^*||\mathbf{EF}^*_{\mathbf{M}*}$.and $\mathbf{M}^*||\mathbf{EF}^*$.by Assumption 3.1. Hence, it is contradiction that $\mathbf{M}^*$ is the best model in this SMP.

This proposition shows that the best solution for SMP should be a model and its full experimental frame. In other words, if an experimental frame is a superset or a subset of the full frame of the best model, it cannot be the best for SMP. It means the experimental frame should fit well to the best model without any unnecessary redundant data to achieve modeling objectives. Hence, if the objectives of simulation modeling are changed, the previous model and its full frame should also be changed to satisfy the new objectives.

3.1.4 Analysis of SMP with Multiple Sets of Objectives

For a single **RS**, since there can be many simulation projects with various objectives, there are needs for many models and experimental frames. Suppose that there are multiple sets of objectives denoted by $\mathbf{Obj^1}$, $\mathbf{Obj^2}$, …, $\mathbf{Obj^K}$ for a real system (**RS**), and let $\mathbf{Obj}^+ = \bigcup_{i=1}^{K} \mathbf{Obj^i}$. In addition, let the pair ($\mathbf{M}^*$, $\mathbf{EF}^*_{\mathbf{M}*}$) be an optimal model and experimental frame for a SMP with $\mathbf{Obj}^+$. Then, $\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{d} \mathbf{Obj^i}$ for ∀i, because $\mathbf{Obj}^+ = \bigcup_{i=1}^{K} \mathbf{Obj^i}$ and $\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{d} \mathbf{Obj}^+$. Let $\mathbf{M}^*_{\mathbf{i}}$ be an optimal model for the SMP with $\mathbf{Obj^i}$, and then its corresponding full frame, $\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$ is the optimal frame by the Proposition 3.1.

[Proposition 3.2] Higher Fidelity Leads to Higher Reusability

If $\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}} \xrightarrow{a} \mathbf{M}^*$, then $\mathbf{M}^*_{\mathbf{i}}||\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}} \equiv \mathbf{M}^*||\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$, and $\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{d} \mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$

Proof:

Since $\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{a} \mathbf{M}^*$ and $\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}} \xrightarrow{a} \mathbf{M}^*$, Fidelity($\mathbf{M}^*_{\mathbf{i}}||\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$) = Fidelity($\mathbf{M}^*||\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$) from the definition of relative fidelity (Definition 2.1). Hence, by the equivalence case of Assumption 3.1, $\mathbf{M}^*||\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$ is equivalent to $\mathbf{M}^*_{\mathbf{i}}||\mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$, and $\mathbf{EF}^*_{\mathbf{M}*} \xrightarrow{d} \mathbf{EF}^*_{\mathbf{M}*\mathbf{i}}$ by the definition of full frame of model.

This proposition is illustrated graphically in Figure 3.1.

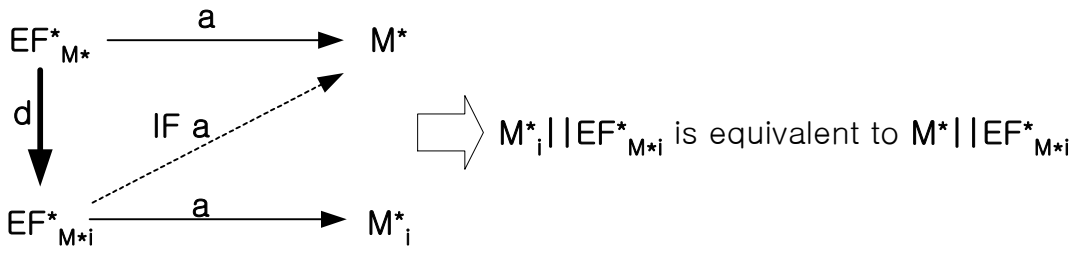


Figure 3.1 Graphical illustration of Proposition 3.2

It means if $\mathbf{EF}^*_{\mathbf{M*i}}$ is applicable to $\mathbf{M}^*$, then $\mathbf{M}^*$ can be used under experimental frame $\mathbf{EF}^*_{\mathbf{M*i}}$ to get the same simulation results, instead of using $\mathbf{M}^*_{\mathbf{i}}$. Hence, $\mathbf{M}^*$ has a potential to be used for various experimental conditions, and building such a model is important to improve the simulation modeling productivity. In other words, higher fidelity model, $\mathbf{M}^*$ can be a superseding model covering the roles of multiple lower fidelity $\mathbf{M}^*_{\mathbf{i}}$'s. In addition, since $\mathbf{EF}^*_{\mathbf{M*}} \xrightarrow{d} \mathbf{EF}^*_{\mathbf{M*i}}$, we can reuse the detailed experimental frame, $\mathbf{EF}^*_{\mathbf{M*}}$ to derive for various experimental conditions represented by $\mathbf{EF}^*_{\mathbf{M*i}}$'s. It means that detailed experimental frame increases *data reusability*. Data observation and gathering are also expensive because it interrupts the normal system operation. If we use high fidelity model, then the data set gathered by its full experimental frame can be reused for other simulation studies for various objectives. From this point of view, the important thing is how to build $\mathbf{M}^*$ to which various experimental frames $\mathbf{EF}^*_{\mathbf{M*i}}$ can be applicable to reuse $\mathbf{M}^*$ for multiple purposes. It means that the high fidelity model should be easily abstracted for accommodating various experimental conditions.

3.2 The Concept of Virtual Factory

3.2.1 Classification of Simulation Models

For a given real system (**RS**), the total set of models that can be built is denoted by $\mathbf{M}_{\mathbf{RS}}$. Given that modeling is based on the finite number of observable variables in **RS**, the total number of conceptual models for the **RS** is also finite. According to the relations of relative fidelity of models, these models are classified in the form of a *layered directed graph* and using the following abstractable relation.

[Definition 3.2] Abstractable Relation

For a given **RS**, the total set of models in **RS** is denoted by $\mathbf{M_{RS}}$. Then, for two models, $\mathbf{M_1}$ and $\mathbf{M_2}$ in $\mathbf{M_{RS}}$, if $\mathbf{EF^*_{M1}} \xrightarrow{d} \mathbf{EF^*_{M2}}$, there is an abstractable relation from $\mathbf{M_1}$ to $\mathbf{M_2}$, and denoted by $\mathbf{M_1} \xrightarrow{abs} \mathbf{M_2}$.

Since abstractable relation is based on the derivability relation between full experimental frames of models, it satisfies the following conditions:

Reflective: $\mathbf{M_i} \xrightarrow{abs} \mathbf{M_i}$

Transitive: $\mathbf{M_i} \xrightarrow{abs} \mathbf{M_j}$, and $\mathbf{M_j} \xrightarrow{abs} \mathbf{M_k}$, then $\mathbf{M_i} \xrightarrow{abs} \mathbf{M_k}$

Asymmetric: $\mathbf{M_i} \xrightarrow{abs} \mathbf{M_j}$, but $\mathbf{M_j} \xrightarrow{\sim abs} \mathbf{M_i}$

Using the above conditions, for a model, $\mathbf{M} \in \mathbf{M_{RS}}$, we can define *abstractable class*. Abstractable class of **M** is a set of models that can be abstractable from the model **M**.

$$[\mathbf{M}]=\{m \mid \mathbf{M} \xrightarrow{abs} m,\ m \in \mathbf{M_{RS}}\}$$

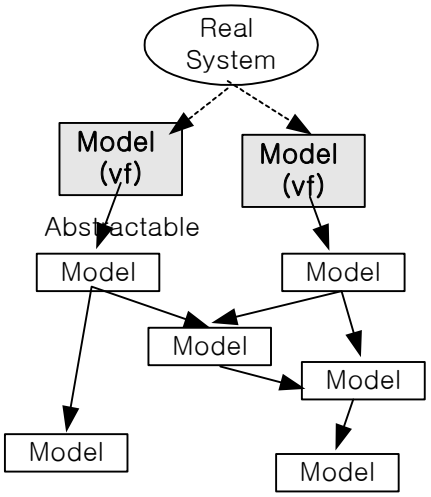


Figure 3.2 Illustration of abstractable relations for a $\mathbf{M_{RS}}$

Among models in an abstractable class of **M**, because the abstractable relation is defined by the derivability relation, the model **M** has the highest fidelity among the models in the class. Figure 3.2 illustrates a layered directed graph showing abstractable relations among the models in $\mathbf{M_{RS}}$. An arc represents an abstractable relation between two models, and any reachable model from a model **M** is included in [**M**]. Since there is no *directed cycle* in this graph, models in a path can be ranked by the level of fidelity.

These models in $\mathbf{M_{RS}}$ can be divided into two groups: those that can be abstractable from other models, and those that cannot be abstractable from any other model in $\mathbf{M_{RS}}$. A model in the later case is called a *representative model* for **RS** and defined formally as follows:

Representative Model: A model (**vf**) in $\mathbf{M_{RS}}$, is a representative model for **RS**, if $\mathbf{vf} \notin [\mathbf{m}]$, for any $\mathbf{m} \in \mathbf{M_{RS}} \backslash \{\mathbf{vf}\}$.

As it is illustrated in Figure 3.2, the representative models are placed on the top of the abstractable model graph, and from these models, all other models can be abstracted. Hence, if we have whole set of representative models defined for a **RS**, then all models in $\mathbf{M_{RS}}$ can be derived from one of the representative models with an abstractable relation. The entire set of the representative models is called *virtual factory* and denoted by **VF** as follows:

$$\mathbf{VF} = \{\mathbf{vf} | \ \mathbf{vf} \notin [\mathbf{m}], \text{ for any } \mathbf{m} \in \mathbf{M_{RS}} \backslash \{\mathbf{vf}\}\}$$

3.2.2 Classification of Model Abstraction Methods

In the Virtual Factory Approach, abstraction is a key concept to build simulation models from a high fidelity model. Abstraction is the process underlying model construction whereby a relatively sparse set of entities and relationships is extracted from a complex reality. The complexity of a model can be taken as the "product" of its scope and resolution. *Scope* refers to the breadth of cover in the real world; *resolution* refers to the depth or the number of variables in the model. Using various model abstraction methods, we can reduce the complexity by reducing the scope of a model or its resolution (Zeigler 1998). Zeigler *et al*. (2000) classified types of abstraction methods that are used in simulation modeling widely. The classification of abstraction methods is presented in Table 3.1.

Table 3.1 Classification of Abstraction Methods in Simulation (Zeigler *et al.* 2000)

Abstraction Type	Brief Description	Affects Primarily
Aggregation	Combining groups of entities into a single entity which represents their combined behavior when interacting with other groups	Resolution
Omission	Leaving out entities, variables or interactions	Scope
Deterministic/Stochastic Replacement		Resolution
Deterministic-> Stochastic	Replacing deterministic descriptions by stochastic ones, can result in reduced complexity when algorithms taking many factors into account are replaced by samples from Easy-to-compute distribution	
Stochastic->Deterministic	Replacing stochastic descriptions by deterministic ones, e.g., replacing a distribution whose values are sampled by a constant value which is the mean of that distribution	

In the simulation modeling framework, a simulation model (**M**||**EF**) consists of model structure **M** and experimental frame **EF**. Hence, we can also classify abstraction methods according to the model structure or experimental frame. While abstracting **EF** involves manipulating observed data, abstracting **M** requires modifying the internal structure of the model. Hence, if possible, it is more economical to use **EF** abstracting methods than **M** abstracting methods. In the following, we explain and classify abstraction methods in the simulation modeling framework.

EF Abstraction Methods

Merge Operation ($\oplus$)

Merge operation is to find a union of the contents of variables without differentiating.

$$A \oplus B \oplus \cdots \oplus N = \{x | x \in \text{Contents}(A) \cup \text{Contents}(B) \cup \cdots \cup \text{Contents}(N)\},$$

where A, B, ···, N are observable variable in **RS**

For example, suppose that there are two product types, of which processing times can be observed separately with PT_a and PT_b. Then, the types of product are abstracted (or ignored) in modeling by merging the observed values of PT_a and PT_b.

Add Operation (+)

Add operation is to add observed values of variables, which are sequentially correlated.

$$A+B+ \cdots + N = \{x | x = a+b+ \cdots + n, a \in \text{Contents}(A), b \in \text{Contents}(B), \cdots ,$$

and $n \in$ Contents(N) with respect to a same entity}

For example, suppose that X is an observable variable for setup time, and Y is an observable variable for processing time. Then, these two variables can be abstracted in modeling by adding setup and processing time as one observable variable.

Nullify Operation (∅)

Nullify operation is a unary operation, which does not take an observable variable into account in modeling. From the above example, if we ignore the setup time in modeling, observable variable X is nullified.

M Abstraction Methods

Randomized Control

This abstraction method is to replace a complex logical rule or algorithm by statistical distribution based on observations. For example, part release can be controlled by other status conditions depending on the algorithm used. In the case of CONWIP control, the algorithm checks current level of WIP, and if it is below the predetermined constant WIP level, the algorithm releases a new product into the factory; otherwise, it does not release a new product. It is not easy to implement such an algorithm in simulation modeling. However, the complex implementation can be replaced by a statistical distribution estimated from the observed values of variables. After observing inter arrival times, we can estimate statistical distribution and parameters that characterize product inter arrival behaviors. Since a computer program can easily generate samples of a statistical distribution, randomized control method can easily be implemented in simulation modeling.

State Space Reduction

State space reduction method is to reduce the number of event handling procedures by reducing the number of states of a variable. For example, suppose a state variable, **S** has three states, S_1, S_2, and S_3. For incoming events, depending on the current state of the variable, the simulation code selects different event handling procedures. If we can group these states into one, we can eliminate the state variable and reduce the procedures to maintain states of variable explicitly in modeling. Instead, we can use frequency data that can be observed by experiment. After observing frequency of state variable **S**, simulation can generate current state according to the state frequency.

3.2.3 Modeling Economics in Virtual Factory Approach

The Virtual Factory Approach is to build a simulation model from a virtual factory, which represents a system with high fidelity, by model abstraction. Compared to a modeling methodology to build a minimum model from scratch, the Virtual Factory Approach uses an exiting high fidelity model. However, building a virtual factory can be a major undertaking. The comparison is analogous to the trade-offs between setup and incremental cost. The following is a simple economic model to explain the benefits of the Virtual Factory Approach.

For a given **RS**, let a corresponding virtual factory be denoted by **vf**. In the Virtual Factory Approach, to build a model for a set of objectives, $\mathbf{Obj_i}$, we can abstract **vf** to achieve the $\mathbf{Obj_i}$, which procedure is denoted by Abstract($\mathbf{vf}|\mathbf{Obj_i}$) = $\mathbf{A_i}$. In case of building a model from scratch, the modeling procedure is denoted by Modeling($\mathbf{Obj_i}|\mathbf{RS}$)

= $\mathbf{M_i}$. Hence, for a series of simulation studies, the Virtual Factory Approach is preferred if

$$\text{Cost}(\mathbf{vf}) + \sum_{i=1}^{n} \text{Cost}(\mathbf{A_i}) \leq \sum_{i=1}^{n} \text{Cost}(\mathbf{M_i}),$$

where Cost(Q) is a cost function to build model Q. Suppose that Cost($\mathbf{A_i}$) = **a**, and Cost($\mathbf{M_i}$) = **m**, for all i, and Cost(**vf**) = $\mathbf{m}^*$, if n > $\mathbf{m}^*$/(**m-a**), then the Virtual Factory Approach is preferred

We can see from the above equation that as a high fidelity model can be built rather inexpensively and the costs for abstraction are smaller, the Virtual Factory Approach can be economically justified. As we mentioned in the previous section, in general, applying abstraction techniques is much simpler than building up a model. Hence, it is a critical success factor in the Virtual Factory Approach to develop economical methodology for building a virtual factory in which modeling structure can be easily abstracted for various objectives.

3.3 Summary

The Virtual Factory Approach, for a given real system and modeling objectives, is to build an executable simulation model by abstracting from an existing high fidelity simulation model (called virtual factory). Because a virtual factory contains a lot of system knowledge, we can systematically reuse the implemented system knowledge imbedded in the virtual factory by abstracting some of the logical rules. If we have a well-designed virtual factory, we can easily derive many abstracted models from it.

In order to apply this approach to simulation modeling, we need theoretical foundations to validate the correctness of this approach. That is, the model abstracted from a virtual factory should be equivalent to models built from scratch for the same modeling purpose. However, it is very difficult to determine whether two simulation models are interchangeable or not without actually running both models under all possible experimental conditions and comparing their output results.

Therefore, we have defined a relative fidelity indicator under a formal simulation modeling framework as a surrogate method to determine the equivalence of simulation model without actual experiment. Based on the relative fidelity indicator and the simulation modeling framework, we formulated a simulation modeling problem, and derived some of the properties of simulation modeling and fidelity. The first property is that the higher fidelity model has more reusability. It means that if we have a well-designed virtual factory for a target system, we can derive many abstracted models for achieving various simulation objectives from the virtual factory. Under same experiment frame in particular, if two models are applicable, i.e. if the models can be experimented under the same experimental frame, these two models are equivalent in terms of relative fidelity.

In this chapter, we also classified model abstraction methods that have been widely used in simulation modeling into two categories. One is to abstract observable variables in experiment frames, and the other is to abstract logical rules in model structures. In general, abstracting an experimental frame is easily implemented since it just deals with

data manipulation for input and output observable variables without changing the programming structure of the model. Since a virtual factory consists of a lot of input and output interface variables interacting with the experimental frame, we can derive many abstracted models with simply applying experimental frame abstraction methods.

Since the Virtual Factory Approach is based on a high fidelity model, it is critical to develop a virtual factory economically, which can be applicable to various experimental frames. In practice, there are difficulties to build high fidelity models for general systems. However, if we confine interesting systems to a specific domain, there is room for developing a modeling methodology for building a high fidelity model efficiently. In the following chapters, we present a modeling methodology called the *Reference Model Approach* to build high fidelity simulation models for systems included in a specific domain.

CHAPTER IV

SIMULATION MODELING METHODOLOGIES

4.1 Introduction

Simulation modeling generally follows three-steps: 1) analyzing the target system, 2) conceptual modeling, and 3) implementing the computerized model. Although it is a simple methodology, these three steps have been a basic guideline for simulation modeling. Other research areas of simulation such as input data modeling (Banks *et al*. 1996, Johnson and Mollaghasemi 1994, Law and Kelton 1999, Vencent 1998), output data analysis (Alexopoulos and Seila 1998, Kleijnen 1975, Law and Kelton 1999), design of experiments (Kleijnen 1998, Neter *et al*. 1990), and various statistical validation and verification techniques (Balci 1998, Sargent 1996) have been widely studied and utilized in practice. However, as Willemain (1994, 1995) pointed out, modeling is one of the least understood elements of simulation methodology, and there are few systematic modeling methodologies generally accepted by simulation practitioners.

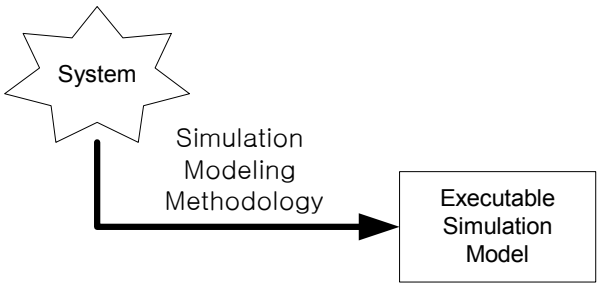


Figure 4.1 A Simulation Modeling Methodology

Within the basic three-step procedure, modelers usually rely on simulation software tools and their modeling experiences to build simulation models. Model designers analyze systems with the help of software support, and translate the results of analysis to simulation constructs that are executable on the simulation execution platform. Hence, the modeling procedure and resulting models heavily depend on the modeler's knowledge of the simulation tool and experiences in modeling. For these reasons, modeling procedure is individualized in an *ad-hoc* way, and there can be significant differences in the output results of simulation depending on modelers.

Another problem in *ad-hoc* modeling procedures is that the results of simulation modeling such as the system knowledge acquired from system analysis and the simulation models are not systematically reusable and shareable. This is because the simulation modeling is customized only for a given system. In order to reuse simulation models systematically, the models should be designed for reuse on purpose.

In this chapter, we present the state of the industry in simulation modeling, and point out the problems of the current methodologies. Then, we propose a new simulation modeling methodology to address the problems.

4.2 Current Simulation Modeling Methodologies

There are many ways to build simulation models. In this section, we classify them by the types of modeling tools as illustrated in Figure 4.2, and explain the features of modeling methodologies in each category.

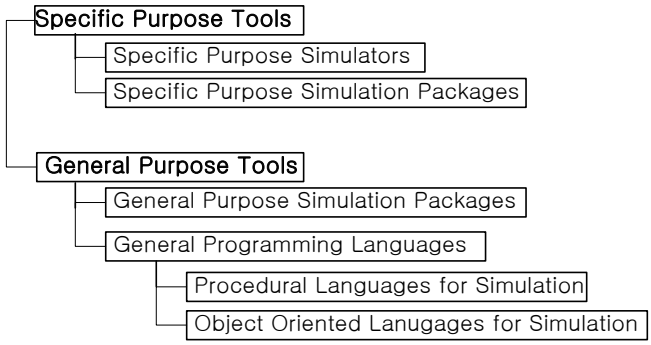


Figure 4.2 A Classification of Simulation Tools

Specific Purpose Simulators

A simulator is a computer model to simulate a specific type of system by specifying parameter values in the model. Hence, if there exists a simulator satisfying the requirements of a simulation project, we can simply use it to get the simulation results by specifying parameters. Regarding model reuse, a simulator has a high level of model reusability, but it can be applied only for extremely limited domain of systems.

Simulation Software Packages

We can separate the variety of simulation packages into two categories. One of these is general-purpose software, and other is the software for specific application domains such as manufacturing, telecommunication, and business systems. Currently, there are many commercial simulation software packages, and the features of main software are described in detail in Banks (1998).

Simulation packages are specially developed modeling systems to support simulation modeling and execution. Whether they are general-purpose or special-purpose tools, they

provide a set of building blocks and modeling rules to facilitate the simulation modeling process. However, while the simulation packages provide convenience in simulation modeling for a certain range of modeling fidelity, it is very difficult to model complex behavior not supported by basic building blocks. Although most of simulation packages provide add-on functions to link external program codes, add-on user codes can break the structure of simulation models and make it hard to manage simulation models for upgrading and reusing.

General Programming Languages

Instead of using simulation packages, we can build simulation models by using general programming languages such as FORTRAN, C/C++, Java, and so on. Usually, these programming languages are used together with simulation execution libraries that provide the basic functions for managing the future event queue and global simulation time. Since they have high flexibility in programming, the programming languages enable high fidelity modeling. However, without well-organized programming structure, it takes a longer time to build simulation models and it requires higher level of programming skills than simulation packages. Because of these reasons, it is important to design program codes easily reusable for reducing program development time and costs.

Object-Oriented Simulation

In the 1990s, the development of Object-Oriented Programming (OOP) (Budd 2001) affected simulation modeling as it did other computer science fields. Because it represents entities in a real system with a special programming structure called *Class*,

OOP enables simulation modelers to build high fidelity models and to reuse program codes encapsulated in a Class. It is therefore expected that we could accumulate developed programming components in the component library, which are reusable for new system modeling.

4.3 Problems of Current Modeling Methodologies

There is no perfect simulation modeling methodology that satisfies all criteria. However, it is meaningful to analyze the weak points of current modeling methodologies in preparation for developing a new modeling methodology to overcome these weak points. In this section, we analyze intrinsic problems in the current methodologies.

4.3.1 Lack of Simulation Model Reuse

The reuse of existing models can significantly reduce the development time and costs, and improve the quality of newly developed simulation models. However, as Overstreet *et al*. (2002) mentioned, reusing existing models is difficult in practice, and although it is the focus of much research in the simulation community, they believe that improving model reuse is a *grand challenge* in simulation.

Similar to general software reuse, there are two important criteria in simulation model reuse, i) modularity and ii) independence. Modularity refers to the property of a model that has been decomposed into a set of cohesive modules. Independence refers to the degree that a module can be used without having relationship with other modules. Therefore, it is important in designing modules that each module should have strong

internal cohesion but weak external coupling with other modules to increase the usability in various environments (Stevens and Pooley 2000).

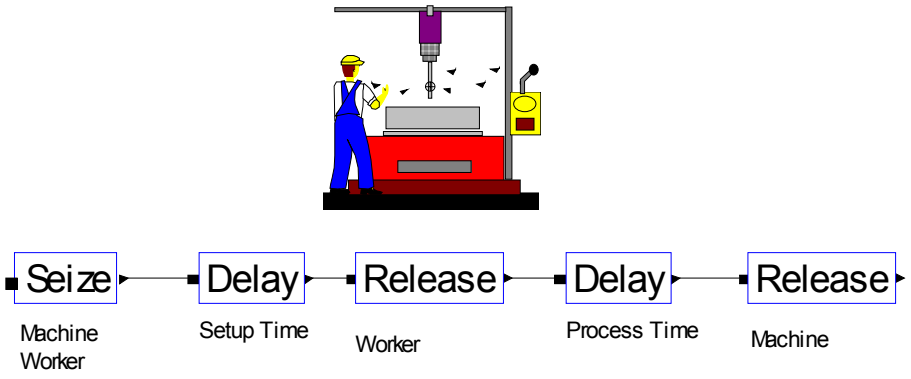


Figure 4.3 Illustration of Modularity in Process-Oriented Modeling

Applying these two criteria, simulation modeling based on the process-oriented modeling abstractions has difficulties in the simulation model reuse. Process-oriented simulation maps system behaviors as a process of entity flows. Each process consists of activities of various objects and each object is used in multiple processes. Therefore, it is difficult to modularize based on either object or process. As an example, suppose there is a manufacturing system shown in Figure 4.3. The manufacturing system consists of two resources, a machine and a worker, who tends to this and other machines. Under process-oriented modeling, the behavior of the system is represented as a process of material flow using the machine and the worker. Since the process consists of interacting behavior of the machine and the worker, it is difficult to modularize the process into reusable modules in the form of object units. On the other hand, the process itself can be a unit of reusable modules. However, since the worker object is used for multiple processes, the

change of worker's behavior in other processes can influence the current process. This strong external dependency makes it difficult to reuse the modules in other environments.

The object-oriented simulation modeling using OOP has received attention as a potential tool to resolve the model reuse problem (Adiga and Glassey 1991). Since a *Class* is a blueprint or prototype that defines attributes and behaviors, reusable objects can be encapsulated naturally by mapping with objects of the target system. However, although it provides a programming structure to encapsulate program codes for facilitating model reuse, it is not that OOP language itself guarantees the independency of reusable objects. If a reusable object is dependent on other components outside of the object, it is unavoidable to modify program codes inside of reusable objects, whenever it is used with different outside components.

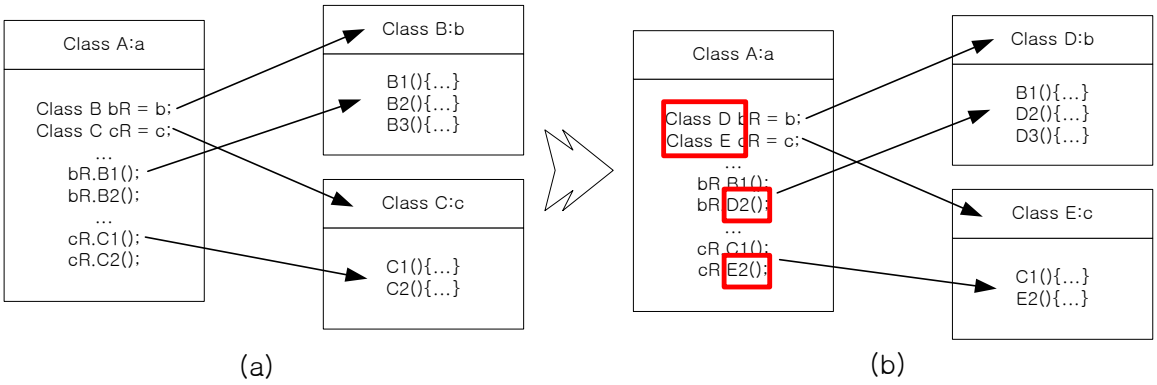


Figure 4.4 Environment Dependency of Object-Oriented Programming

Figure 4.4 illustrates dependency problems of object modeling. In modeling (a), the `object a` of `Class A` has reference pointers, `bR` and `cR`, for `object b` of `Class B` and `object c` of `Class C` respectively. When we try to reuse `object a` of `Class`

`A` for different modeling (b), since the environment has changed, the adjustment for linking new environment is necessary. Hence, some of the class names and behaviors have to be changed in `Class A` for reusing the object in a new environment. When we design classes without considering future model reuse, this kind of modification should be required in many situations. As a practical example, we can consider a machine operating in a factory A. Suppose that the same type of machine is operating in a different factory B. Then, since the machine type is same and its functions are equivalent, the machine object should be reusable conceptually for both factories. However, because of the differences of environments (factory A and B) using the machine object, additional programming for reuse is inevitable.

Therefore, in order to reduce the dependency of objects, the Class should use standard reference pointers that can link with objects in various environments. However, in this case, the simulation modeler should consider all possible environments that can link with the object in the future, but it is not easy to design a general class structure, and it is also a costly way to achieve model reuse. Indeed, to maximize the benefits of model reuse, it is necessary to define a specific boundary within which the reusable objects can be formalized efficiently.

4.3.2 Lack of Flexibility in Modeling Fidelity

One of desirable features in simulation modeling tools is that it should have a capability to model a real system with various abstraction levels. This modeling flexibility supported by simulation modeling tools is an important evaluation criterion. In the case

of a simulator, it supports fixed model fidelity. Hence, the application domain of a simulator is restricted to a specific type of system with fixed model fidelity.

Usually, modelers would expect that general-purpose simulation packages enable them to build simulation models with various levels of fidelity according to the modeling objectives. In reality, the fidelity of model built with a simulation package is restricted by the structures of building blocks and modeling syntax. Although most packages provide add-on functions to link with external programs, it is difficult to link exceptional codes with consistent programming and efficient maintenance. Hence, when a system is given with modeling objectives, it is important to select a simulation package that can provide sufficient modeling fidelity for the given problem.

The maximum fidelity supported by a simulation tool depends on the design of the building blocks. As a set of building blocks are designed to enable modelers to model in more detailed level, it becomes closer to the programming language to model complex behavior, so it can lose the benefits of convenience as a software package. Therefore, in practice, the software vendors provide various application templates running on top of the basic building blocks.

4.3.3 Lack of Consistency in Modeling

Since current simulation modeling methodologies are based on the available simulation modeling tools, simulation models and the results of system analysis can be different

according to the level of the modeler's expertise in using modeling tools and modeler's experiences.

The inconsistency occurs when modelers select different options in modeling. In modeling a system, there are several views to model a real system. These modeling views are different according to the level of the modeler's expertise in the simulation modeling tool and modeling experiences. Even though these modeling views can be implemented correctly as executable simulation models, their simulation results may not be equivalent. Therefore, in order to reduce the inconsistency, unnecessary modeling options should be minimized. However, in simulation modeling for general systems, it is difficult to check systematically whether a modeling option is necessary or not for a given problem.

4.3.4 Unrealistic Control Abstraction

Since the 1990's, research on object-oriented simulation for manufacturing systems (Adiga and Glassey 1991, Mize and Pratt 1991, Narayanan 1994, Zeigler 1991) have mentioned the unrealistic control abstraction problem to criticize the process-oriented simulation modeling conventions of commercial simulation packages.

Manufacturing systems consist of two main systems, physical systems and logical control systems. While physical systems have received more attention for more realistic modeling, logical control systems are abstracted with simplistic dispatching or control rules. This is because the logical control systems often involve many objects with complicated interactions, and it is far more difficult to formalize them with standard

patterns. However, as the modern manufacturing systems are integrated and automated with computer control systems, the logical control systems have a big influence on the performance of manufacturing systems. Hence, we need modeling methodology to model both physical and logical system with high fidelity.

4.4 A Proposed Simulation Modeling Methodology Based on Reference Model

The current modeling methodologies deal with a specific target system. Since the system analysis is focusing only on a given system, the results of modeling one system are not easy to use systematically for modeling other similar systems. However, if we expand our focus on a group of similar systems, we can build a common framework to describe all systems in the group. If the framework has a formal structure and any system in the group can be described in terms of the formal structure, we can generate simulation models automatically from the formal descriptions.

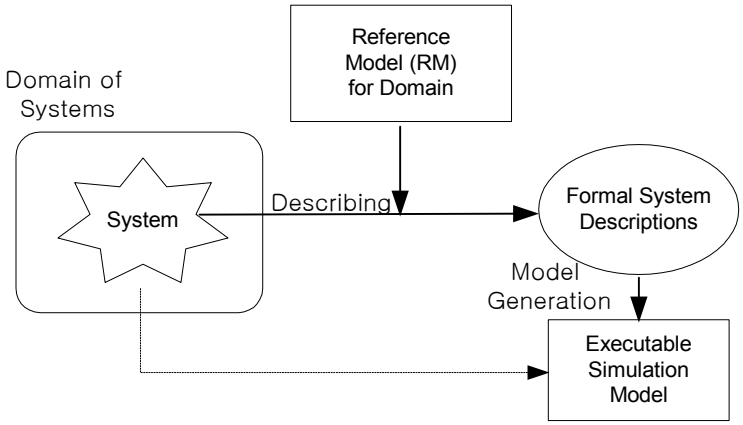


Figure 4.5 A Simulation Modeling Methodology based on Reference Model

We propose a simulation modeling methodology for a group of similar systems based on formal framework. Figure 4.5 shows the modeling procedures. This modeling methodology is neither for a single target system nor for general systems. Instead, it addresses a group of systems that have similar characteristics. This kind of group is called a *domain*. Based on the domain knowledge codified through domain analysis, we can build a formal framework to describe systems in a domain. This formal framework is called a *reference model* (Bodner and McGinnis 2002). In software engineering, domain analysis is used in the domain of software applications (Arango and Prieto-Diaz 1991). Since we can describe any system of the domain in terms of the reference model, we can design a procedure to generate specific simulation models automatically from the formal descriptions. This modeling approach is called *Simulation Modeling Methodology Based on Reference Model*.

The features of the simulation modeling methodology based on the reference model are described as follows:

- This methodology is for the systems in a specific *domain*.
- This methodology is based on the results of *domain analysis* to identify common *modeling entities* that characterize the behavior of systems in a domain.
- *Reference model* of a domain is based on the modeling entities, and is a formal structure for describing systems in the domain.
- Any system belonging to a domain can be described in the structure of reference model, and this description is called a *reference model instance*.

- A *model generation procedure* can be developed to build executable simulation models from the reference model instance.

4.5 Benefits of Reference Model Methodology

Compared to conventional modeling methodologies, the reference model based simulation modeling methodology (in short, *Reference Model Methodology*) has the following benefits in simulation modeling.

Systematic Reuse of Domain Knowledge

A *reference model* is a formal structure to describe the systems in a specific domain, and a *model generation procedure* is a process to convert formal descriptions to executable simulation models. In order to build a reference model and its corresponding model generation procedure for a specific domain, the domain should be exhaustively analyzed, and common domain knowledge should be codified. Finally, the reference model and its model generation procedure represent an integrated structure imbedding the results of domain analysis and common domain knowledge. In other words, in the Reference Model Methodology, describing a target system in the form of a reference model means that we utilize the results of domain analysis and domain knowledge systematically.

Consistency in Simulation Modeling

The Reference Model Methodology reduces or eliminates inconsistencies from two phases: the first one comes from system analysis to get the conceptual model, and the second phase comes from the model generation from the conceptual model. The first

phase is related to modelers' experience, and the second phase is related to modelers' expertise of simulation tools. Since, in the Reference Model Methodology, the system analysis phase is formalized in a reference model, and the simulation model is generated automatically by the model generation procedure, this methodology demonstrates more consistency in simulation modeling than conventional methodologies.

End-User Simulation Modeling Environment

The modeling activities in the Reference Model Methodology consist of three types: the first one is to build a reference model for a specific domain, the second one is to design an automatic procedure to generate executable simulation models, and the third one is to describe a target system in terms of the reference model. The required knowledge or expertise for performing each of these activities is different from the other.

To build reference model instances, the knowledge about the target system is required to decide which factors are important in modeling. After deciding the modeling factors, because the modeling procedure is simple–just filling in the data identified in the reference model– and the descriptions are converted to simulation codes by the model generation procedure, modelers are not required to have a high level of simulation modeling knowledge. On the other hand, the implementer of the model generation procedure do not need to know domain system knowledge well, because their role is to build a procedure to generate simulation codes from a predetermined formal structure. They are required to have both software engineering skills and simulation programming knowledge for this task. However, for the people who design a reference model, they

should have both domain system knowledge and simulation modeling knowledge to build a reference model for a specific domain.

Therefore, if a reference model of a domain and its corresponding model generation procedure are already equipped, system experts can use simulation technology freely without knowing in-depth simulation modeling knowledge (although they should have simulation analysis knowledge, of course) in the proposed modeling methodology. In other words, the Reference Model Methodology enables *end-user simulation modeling* for systems in a specific domain.

Increased Simulation Modeling Productivity

Usually, describing a system is much easier than programming to build a simulation model. In the Reference Model Methodology, since simulation models are generated by automatic procedure from formal descriptions (called reference model instance), once a reference model and its model generation procedure are equipped, then the simulation modeling productivity is highly increased than other methodologies, and its modeling cost is decreased.

Limitations of Reference Model Methodology

However, the Reference Model Methodology has a fundamental limitation: a given reference model is only applicable to a specific domain. One extreme case of this methodology is a simulator. In this case, the set of changeable parameters is a part of a reference model, and providing parameters to the existing simulator is analogous to

generating an instance of simulation model. On the other hand, the other extreme case is the methodology using general-purpose simulation packages. A simulation package provides building blocks for modeling. The structure of the building blocks and their modeling rules can be regarded as a reference model for a system. After building a simulation model using the building blocks, the model should be compiled and converted to the program codes that can be executed in the execution platform of the package. It can be regarded as a model generation procedure. In this case, the process to build a simulation model is not so simple as to describe the system in a reference model. Therefore, to maximize the benefits of the Reference Model Methodology, it is important to decide the scope of domain of which reference model provides efficient modeling.

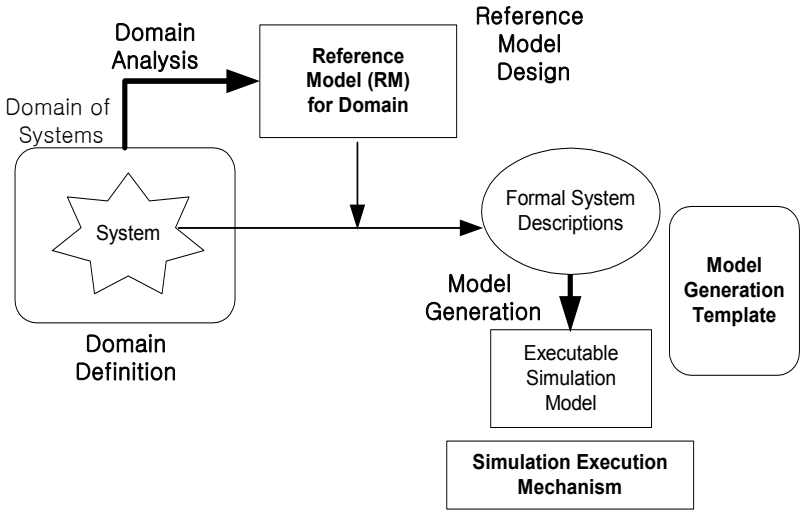


Figure 4.6 Research Tasks for Reference Model based Methodology

4.6 Research Tasks for Reference Model Methodology

For a given domain, two fundamental elements for successful deployment of the Reference Model Methodology are i) reference model and ii) model generation procedure. In order to develop these fundamental elements, the following sequence of tasks should be performed correctly as shown in Figure 4.6.

1) *Domain definition*: Defining the scope of target domain clearly in order to characterize the domain and to provide a standard to determine whether a given system is included in the target domain or not.
2) *Domain analysis*: Analyzing target domain to find common modeling entities and logical rules, which characterize and control the behaviors of systems in the target domain.
3) *Reference model design*: Formalizing the characteristics of modeling entities to describe any system in a domain sufficiently.
4) *Simulation execution mechanism design and implementation*: Building a simulation execution framework that is compatible to execute simulation models generated from formal descriptions.
5) *Model generation procedure development*: Making a mapping procedure from the formal descriptions of a target system and common domain knowledge to generate the simulation models executing within the execution framework.

Although we described the tasks for developing a reference model and a model generation procedure, it is difficult to find general methods to perform each task

effectively and efficiently for any type of domain. Due to the heterogeneous characteristics of each domain, its reference model can have a totally different structure, and it is very difficult to apply common methods for domain analysis. Hence, for a given domain, domain specific methods for performing tasks of the Reference Model Methodology should be developed according to domain characteristics. However, since it is definitely true that a good reference model and a good model generation procedures enable successful deployment of the Reference Model Methodology, task-performing methods should be developed for building a reference model and a model generation procedure to have the following evaluation criteria.

Criteria for evaluating a reference model and a model generation procedure

The reference models can be evaluated by the following criteria:

- *Sufficiency*: A model instance of a reference model should provide sufficient information to build an executable simulation model. Otherwise, the model generation procedure cannot be automated, because it requires additional information with *ad-hoc* formats to supplement the deficiency.
- *Consistency*: For the same modeling objectives, modelers who have the different levels of modeling expertise should build consistent reference model instances. In order to minimize the inconsistency, the reference model should be designed not to provide unnecessary modeling factors that modelers can choose optionally.
- *Reusability*: Model instances of a reference model should be modularized independent of one another. Since a simulation model is generated from the

formal descriptions, it is more important to focus on the reuse of system descriptions instead of program code reuse.

- *Coverability*: A system in a domain can be modeled with the various levels of fidelity depending on the simulation objectives, which is a decision made by modeler. Hence, it is a desirable attribute that a reference model can be applied for various levels of fidelity.
- *Efficiency*: To generate simulation model, the model generation procedure should access the database storing reference model instances. Depending on the structure of reference model, the spending time for model generation can be influenced.

The quality of a model generation procedure can be evaluated by the following criteria.

- *Correctness*: A model generation procedure should generate correct executable simulation models correctly as modelers intended to describe the systems as reference model instances.
- *Simplicity*: Model generation procedure is a process to convert reference model instances to executable simulation models. This converting performance depends on the design of the model generation procedures.

4.7 Summary

In this chapter, we discussed the current simulation modeling methodologies relying on the capabilities of simulation tools. Since these methodologies are strongly dependent on the modeler's expertise of simulation modeling knowledge and experiences, it is hard to reuse and share the results of simulation modeling systematically.

After analyzing the problems of current simulation modeling methodologies, we propose a new simulation modeling methodology based on a reference model. While other methodologies focus on a single target system, the Reference Model Methodology deals with a specific domain of similar systems. This methodology is based on a reference model that is a formal structure for describing systems in the domain and a model generation procedure in which domain knowledge is structured.

In the next two chapters, we apply the proposed methodology to discrete part manufacturing system domain to find a reference model through domain analysis, and propose a model generation procedure corresponding to the reference model.

CHAPTER V

DOMAIN ANALYSIS AND REFERENCE MODEL FOR DISCRETE MANUFACTURING SYSTEMS

In this chapter, we present the concept of the reference model based simulation modeling methodology (in short, *Reference Model Methodology*) for discrete part manufacturing systems. To apply the Reference Model Methodology for a certain domain, we should follow a sequence of tasks: 1) *Domain Definition*, 2) *Domain Analysis*, 3) *Reference Model Development*, and 4) *Simulation Execution Mechanism Design and Implementation* and 5) *Model Generation Procedure*. In this chapter, we deal with the first three tasks related to the reference model, and the remaining tasks for the model generation and execution will be covered in the next chapter.

5.1 Domain Definition

Clear domain definition allows us to determine whether a given system is included in the target domain or not. In this thesis, we develop a reference model and a model generation procedure for the domain of discrete part manufacturing systems. This domain is generic to most of the modern manufacturing systems. Since the main goal of simulation for manufacturing systems is to evaluate system performance for various situations, which depends on time points of material movements, we should focus on material flow in the domain analysis. The domain of discrete part manufacturing systems is described as follows:

Definition 5.1: [Domain of Discrete Parts Manufacturing Systems]

The target domain of discrete parts manufacturing systems is defined as manufacturing systems in which materials flowing through the system are countable objects, and manufacturing operations are controlled by determined logical rules.

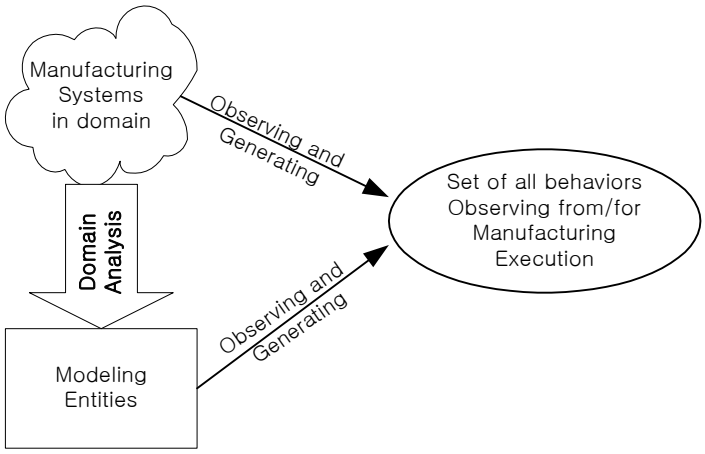


Figure 5.1 The Domain Analysis and The Concept of Modeling Entities

5.2 Domain Analysis

As shown in Figure 5.1, domain analysis is a process to find *modeling entities*, with which we can observe and which generate all observable behaviors of interests. Modeling entities should be defined in the fundamental level to describe all behaviors in the domain sufficiently with their combinations. However, complex subsystems may be modeled more efficiently by using a high level composite modeling entity. Hence, it is key in domain analysis to find a complete and independent set of modeling entities for effective and efficient modeling.

5.2.1 Viewpoint of Domain Analysis

Discrete Event System (DES) (Cassandras *et al*. 1999) concepts provide an effective viewpoint to analyze the discrete parts manufacturing systems. Instead of observing the behaviors of the system continuously, this viewpoint focuses on events that are of interest and important in system analysis. In real systems, observing behavior possibly may have a continuous evolution. However, the viewpoint in DES is not interested in the evolution, but focuses on the starting and the ending points of the evolution. These two points are *events* to characterize the behavior.

However, the challenge in modeling with DES is that events of interest are subjective and different according to modelers and/or the purpose of modeling. Hence, it is important to find a maximum set of event types that fulfill the various purposes of the modeling and at the same time, is not overly complicated.

Two important criteria to determine the maximum set of event types are *observability* and *controllability* of events, which are bounded at the maximum level of fidelity in modeling. Here, observability of an event refers to whether we can perceive the occurrence of the event or not, and controllability refers to whether we can control (i.e. stop or resume) the event or not. Even though an event is interesting and important to understand system behavior, if it is not observable during the system execution, it should not be included in the maximum set of events, and if we cannot control the behavior corresponding to the event, the control event should not be included in the set. For example, when a part enters a machine, if there is no way to observe the behaviors inside of the machine, it is

meaningless to include the related events that occur inside the machine in the maximal set of events. In the event that we cannot stop machining after loading a part until it is finished, the control event to stop the machine process should not be included in the maximal event set. Hence, the analysis of observability and controllability in a domain is a requirement for domain analysis. In discrete part manufacturing systems, material movements and flow controls are important criteria for checking observability and controllability.

5.2.2 Entities in Discrete Part Manufacturing Systems

Entities in manufacturing systems are classified into two distinct groups. One is the *physical entity* group that includes visible entities comprising the physical factory. For example, physical entities are machines, materials, tools, buffers, and transporters. These entities provide the source of data to describe the physical status of system, which is used for decisions about how these physical entities should be controlled to achieve the goals of the manufacturing system. However, these entities have no decision-making functions in themselves. Another entity is the *logical entity* group that includes operation procedures and decision-making related to executing the physical factory. These logical entities can make decisions to move the physical entities based on the status information. Example of such decisions can be WIP movement, resource allocation, equipment maintenance, batching, setups, and other related activities.

The point is that both physical and logical entities are important for the performance of a manufacturing system. It is clear that the capacity and speed of a factory depends on the

capabilities of the physical equipment set. However, it is equally clear that the performance of the manufacturing system is directly impacted by the quality of the logical entities. Therefore, in order to evaluate the performance of manufacturing systems correctly, we need a framework to characterize and describe the *physical entities*, the *logical entities*, and the *interactions* between the two types of entities (Kempf 1996).

5.3 Analysis of Entities in Manufacturing Systems

In this section, we analyze the target domain to identify the physical entities, the logical entities, and their interactions in manufacturing systems sequentially.

5.3.1 Modeling Physical Entities in Manufacturing Systems

From the perspective of material flow, the execution of manufacturing systems consists of three fundamental physical behaviors: *Transformation*, *Transportation*, and *Storage* of materials. We need to identify the classes of physical entities that express these fundamental physical behaviors. Here we identify three entity classes: *Material*, *Location*, and *Transport System*. Together, these three physical entity classes can express all fundamental physical behaviors.

Material

All of these fundamental physical manufacturing behaviors are related to material handling and flow. Hence, *material* is a fundamental physical entity, which is defined as an object to be transformed, transported, or stored during the manufacturing execution. From this definition, materials can be classified into two groups:

- *Part Type Material*: Materials that will be a part of the product. Examples are finished products, subassemblies, and raw materials.
- *Auxiliary Material*: Materials that are not a part of the product. Examples are tools, and fixtures, which are shared and moved in a factory.

In manufacturing execution, these materials can be typically characterized by three attributes: *type*, *status*, and *id* of material, and defined as follows.

> Definition 5.2: [Material]
>
> A Modeling Entity, Material is defined as any object that is transformed, transported, or stored during manufacturing execution, and a material is characterized by three attributes: *type*, *status*, and *id*.

Location

Location is a physical entity type representing a special place where some fundamental manufacturing behavior can occur. From the perspective of the material handling and flow, we classify the locations based on three fundamental manufacturing behaviors: *Transformation*, *Transportation*, and *Storage*. The following is the definitions of these behaviors and their associated locations.

Transformation behavior changes the status of materials. Not only can the changes be physical such as processing, assembling, and separating, but also logical such as *pass* or *fail* in inspection. In transformation, we can use two events, *starting* and *finishing*

transformation, to sufficiently describe the behavior of material handling and flow, and refer this type of locations on which materials are transformed as *Process Port* (PP).

Transportation behavior changes the location of materials. In this physical behavior, there are two related events, *starting* and *finishing transportation*. As soon as material at a location leaves, the event of starting transportation occurs, and when it reaches its destination location, the event of finishing transportation event occurs again. By mapping these two types of event on a singe location, we can define two events: *entering* and *leaving* of a location related to transportation behavior. Let us then call this type of locations *Load Port* (LP).

Storage behavior does not change the status or location of the material. We call this type of storage locations *Buffer* (BF). Since the storage behavior is described by the duration of material staying in a location, Buffer is also a physical entity to describe the storage behavior by *entering* and *leaving* of a group of locations. Compared with Load Port, one thing that is different is that while Load Port has a space for only a single unit of material, Buffer has multiple spaces for storing multiple materials.

> Definition 5.3: [Location]
>
> A Modeling Entity, Location is defined as a physical place where one or more of three fundamental manufacturing behaviors (Transformation, Transportation, or Storage) can occur. Depending on the types of manufacturing behaviors, locations classified into three types, Process Port (PP), Load Port (LP), and Buffer (BF).

Transport System

Transport System (TS) exhibits very complex behavior because it links materials, locations and other resources in a system. A transport system is links locations and transport equipment to enable a unit of materials to move from one location to another. From this point of view, transport system includes any systems that move materials inside equipment.

The characteristics and parameters of the transport system vary significantly with the types of systems. In order to analyze the behavior of transport systems, we need to classify the types of systems explicitly. Since transport systems have different internal behaviors according to their type, their behaviors can be divided into two groups: *common behaviors* of transport systems, and *particular behaviors* corresponding to a specific type. For example, while both a conveyor system and an AGVS (Automated Guided Vehicle System) have common behavior, i.e. to move materials from one location to another, each system has a different mechanism to transport materials.

Transport systems can be classified into the following five groups from the system modeling point of view:

- *Transport System with Guided Path* (`TG`): Transport systems where transporters move along with fixed guided paths or railroads. E.g. RGVS (Railroad Guided Vehicle System), OHT (Overhead Transporter), or Wire-guided AGV.
- *Transport System with Free Path* (`TF`): Transport systems where transporters move without any guided paths or railroads. Transporters can move from one

location to another without traffic congestion. E.g. forklifts, transport workers, robot hand or path free AGVs

- *Conveyor System* (`CS`): Transport systems consisting of conveyor lines. Depending on whether materials on the conveyor are to be accumulated or not, the conveyor systems are divided into synchronized or asynchronized types.
- *Resource Restricted Point-to-Point Transport Model* (`RT`): A model of transport systems without considering the mechanism of how to transport between locations. For example, in a system with two transport workers, if there is material to be moved, the material should wait until one of the transporters is available.
- *Unrestricted Point-to-Point Transport Model* (`UT`): Similar to above, but with no resource constraints. In this transport model, when material is to be moved, the material is moved to the destination without waiting.

Each type of transport system has an associated set of attributes. The following list shows these attributes in accordance with the types of transport systems:

- Transport System with Guided Path (`TG`):

  `<Speed, Length, Acceleration Factor, Deceleration Factor, Weight Factor, Initial Position>`

- Transport System with Free Path (`TF`):

  `<Speed, Initial Position>`

- Conveyor System (`CS`):

  `<Speed, Conveyor Type, Cell Size>`

- Resource Restricted Point-to-Point Transport Model (`RT`):

  `<Speed, Capacity>`

- Unrestricted Point-to-Point Transport Model (UT):

  <Speed>

Hence, for any type of transporters, all attributes can be described with the following unified format.

<Type of Transport System, Speed, Length, Acceleration Factor, Deceleration Factor, Weight Factor, Initial Position, Conveyor Type, Cell Size, Capacity>

Another important component to describe transport systems is *transport network*. It is defined as a directed graph, denoted by TN = (N, A), where N is the set of locations, and A is the set of directed arcs, which represent the possibility of transportation from one location to other locations. Transport system is a composite modeling entity, defined as follows:

Definition 5.4: [Transport System]

A Modeling Entity, Transport System is defined as a composite modeling entity linking locations and transport equipment to move a unit of materials from one location to another. A transport system is characterized by three factors: the type of transport system (TYPE), the specification of transport device (TP), and the transport network (TN) with following formats:

TYPE = [TG|TF|CS|RT|UT], where [a|b] means either a or b,

TP = <Type of Transport System, Speed, Length, Acceleration Factor, Deceleration Factor, Weight Factor, Initial Position, Conveyor Type, Cell Size, Capacity>,

TN = (N, A) where N is the set of locations, and A is the set of directed arcs.

5.3.2 Mapping Behaviors of Physical Entities to Modeling Entities

In this section, we define the behaviors of the modeling entities to map from those of physical entities.

Material

In manufacturing execution, the major concern for materials is about accessing and updating the information of the materials. Manufacturing systems can acquire the information of material by seeing the appearance or scanning the identification code of the material, and get more necessary information by accessing the information system[3] with the observed result or scanned identification code. The structures of information vary drastically depending on each of the manufacturing systems. For example, in some cases, information system includes the process plan for each product type, but in some other flow line systems that have fixed product routing, such information is not required. Practically, it is very difficult to represent all types of information systems in a formal form. However, without considering how the information systems are implemented, we can assume that information related to material can be accessed by the attributes of

[3] Information system does not mean only a computerized system. Rather, it means any system that provides information queried by *id* or *type* of materials.

material: *type*, *status*, and *id*. Hence, a modeling entity, material should have the behaviors to access or to update these attributes.

By using Object-Oriented Modeling (OOM) (Budd 2001), a modeling entity, material is represented as in Figure 5.2 with the attributes and behaviors.

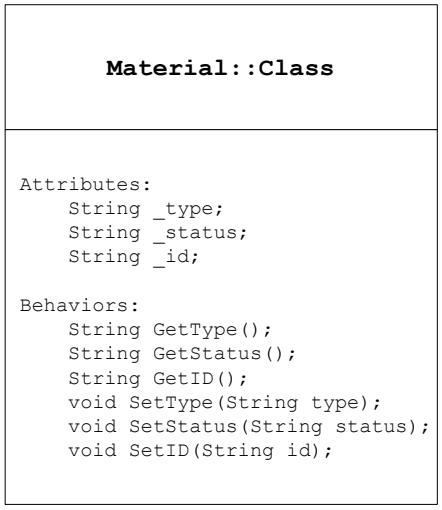


Figure 5.2 Object Model for a Modeling Entity, Material

Location

The behaviors of location can be divided into two groups; common behaviors that all types (Load Port, Process Port, and Buffer) of location have and distinctive behaviors according to the types of location. While *entering* and *leaving* are common behaviors for all types of location, *starting* and *finishing transformation* are distinctive behaviors for Process Port. In addition, for the behaviors to access or to update the state information (which is called *attribute*) of location, PP and BF have different structure. For example, while PP has an attribute about whether a part is in processing on the location or not, BF has a different attribute about how many materials are currently stored in the location.

By using OOM, the common attributes and behaviors of location can be modeled as *interface*, and distinctive attributes and behaviors of specific types of location can be modeled as an *Object* implementing the common interface. In this thesis, the common interface is called *Location Interface*, and the object models of location are shown in Figure 5.3, and detail descriptions of attributes and behaviors are explained in Table 5.1.

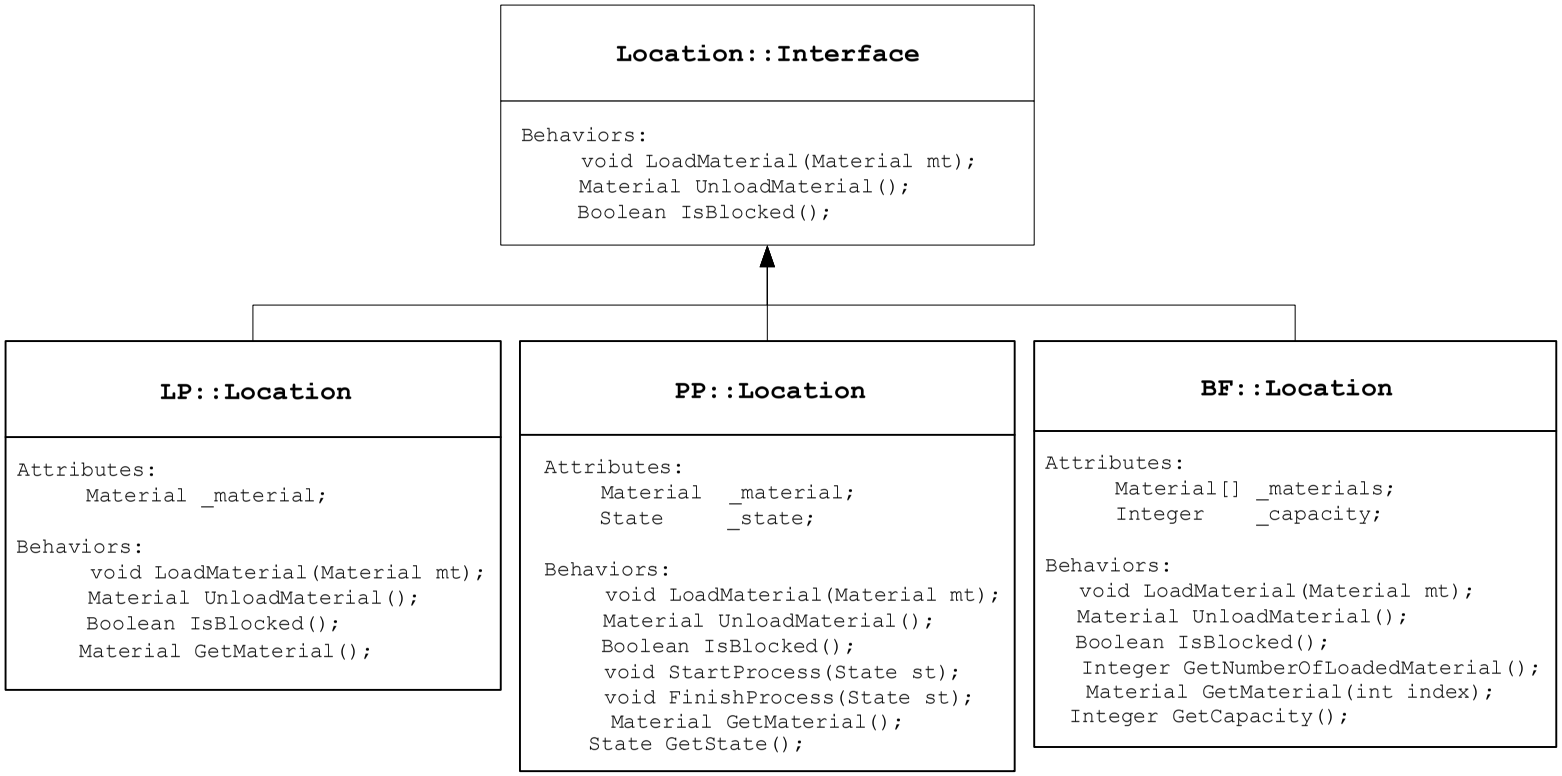


Figure 5.3 Object Model for a Modeling Entity, Location *(LP, PP, BF)*

Table 5.1 Descriptions of Object Model for a Modeling Entity, Location (LP, PP, BF)

Modeling Entity	Attributes	Behaviors
Location ::Interface		• `void LoadMaterial(Material mt):` • `Material UnloadMaterial():` • `Boolean IsBlocked():`
LP ::Location	• `Material _material`: Status information of the material on this LP	• `void LoadMaterial(Material mt):` Put material `mt` on the location. Assign mt to `_material` • `Material UnloadMaterial():` Remove the material on the location and return to material handler. Return `_material` and nullify `_material` • `Boolean IsBlocked():` Check if there is any available space on the location. If `_material` is not null, then return `True`, otherwise, return `False`. • `Material GetMaterial():` Return `_material`.
PP ::Location	• `Material _material`: Status information of the material on this PP • `State _state:` Status information showing the state of this PP.	• `void LoadMaterial(Material mt):` Put material `mt` on the location. Assign mt to `_material` • `Material UnloadMaterial():` Remove the material on the location and return to material handler. Return `_material` and nullify `_material` • `Boolean IsBlocked():` Check if there is any available space on the location. If `_material` is not null, then return `True`, otherwise, return `False`. • `void StartProcess(State st):` Change `_state` to `st`. It performs procedures after starting process. • `void FinishProcess(Status st):` Change `_state` to `st`. It performs procedures after finishing process. • `Material GetMaterial():` Return `_material`. • `State GetState():` Return `_state`.
BF ::Location	• `Material[] _materials`: Status information of the materials on this BF • `Integer _capacity:` Size of buffer	• `void LoadMaterial(Material mt):` Put material `mt` on the last position of the locations. Assign mt to the last position of `_materials[]` • `Material UnloadMaterial():` Remove the first material on the locations and return to material handler. Return the first material in `_materials[]` and rearrange the positions of materials in `_materials[]` • `Boolean IsBlocked():` Check if there is any available space on the locations. If `_materials[]` is full, then return `True`, otherwise, return `False`. • `Integer GetNumberOfLoadedMaterials():` Check how many materials are loaded in this buffer. After calculating loaded spaces in `_materials[]`, return the number. • `Material GetMaterial(int index):` Return `_material[index]` • `Integer GetCapacity():` Return `_capacity`

Transport System

A transport system is described by multiple modeling components: *type of system* (TYPE), *specification of transporter* (TP), and *transport network* (TN). However, such descriptions are not sufficient to build or implement a transport system in real situations, because it requires many and more detailed specifications that are completely different according to types of transport systems.

Although transport systems have distinctive behaviors depending on the types of transport systems, there are standard behaviors that all types have as a transport system, and other specific and unique behaviors that can be encapsulated as abstracted behaviors.

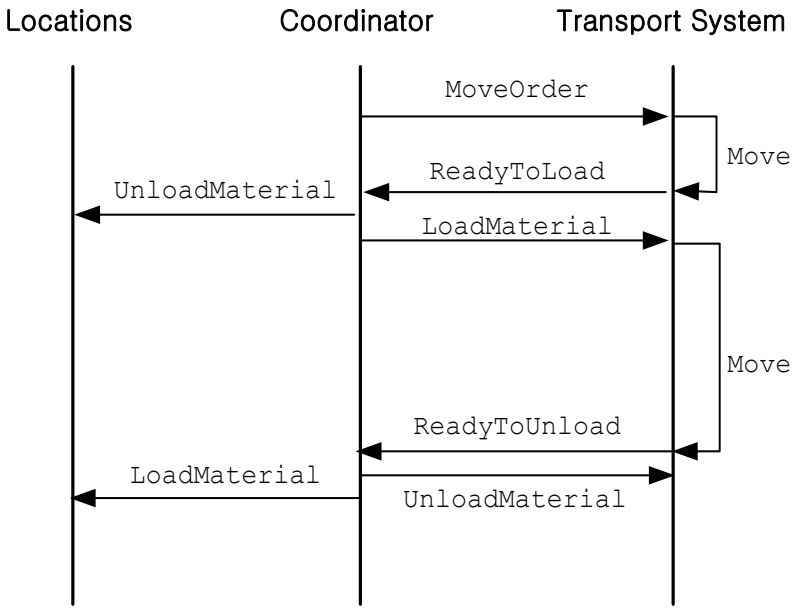
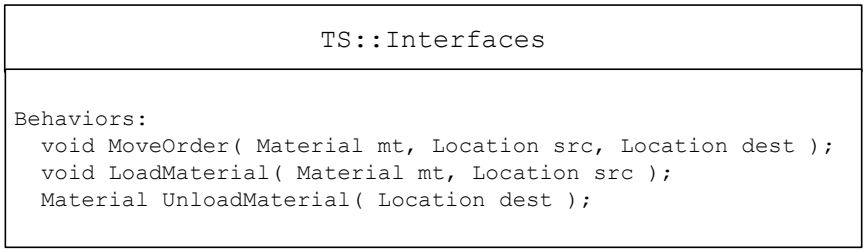


Figure 5.4 Activity Diagram for Standard Behaviors (Interfaces) of Transport Systems

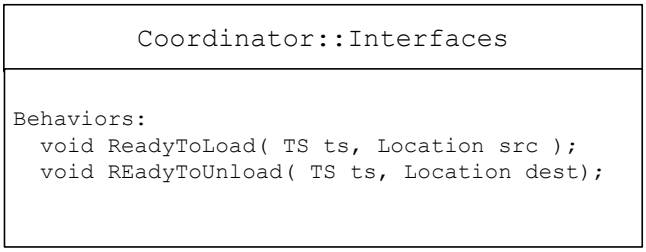
We adopt *Activity Diagram* (Stevens and Pooley 2000) shown in Figure 5.4 to illustrate the standard behaviors that are common for all types of transport systems. In order to move material, first, the coordinator[4] calls `MoveOrder` interface of the transport system. Then, the order is stored in queue, and is in waiting until the transport system is ready to load the material. Internal behaviors of transport systems between `MoveOrder` and `ReadyToLoad` are different depending on the types of transport systems. These behaviors can be simplified as an abstracted behavior called `Move`.

When the transport system is ready to load material, it calls `ReadyToLoad` interface of the coordinator. Then, the coordinator calls `UnloadMaterial` interface of the location, and calls again `LoadMaterial` interface of the transport system with the unloaded material. After being internally processed, when the material has arrived at the destination, the transport system calls `ReadyToUnload` interface of the coordinator. The coordinator, then, calls `UnloadMaterial` interface of the transport system and loads the material by calling `LoadMaterial` interface of the destination location. The standard behaviors (or called *Interfaces*) for transport systems are described in Figure 5.5.

[4] Coordinator is a modeling entity to link physical and logical modeling entities. More detail explanation and formal definition is presented in section 5.5. In this paragraph, the role of coordinator is to interact with transport system and location for moving materials.



(a) Standard Interface of Transport System



(b) Standard Interface of Coordinator

Figure 5.5 Object Model for Standard Interfaces of Transport System (TS) and Coordinator

5.4 Analysis of Logical Entities in Manufacturing Systems

While physical entities are intuitive in modeling, logical entities are less tangible. In general, these logical behaviors are control procedures, rules, and decision-making being applied in manufacturing execution. Because of these formless ambiguities in logical behavior in manufacturing systems, logical entities are difficult to be described formally.

In this section, from these ambiguous controlling behaviors that are hard to formalize, we attempt to separate some parts that can be formally expressed as a logical modeling entity, and analyze logical manufacturing behaviors with the modeling entity. We will refer to the modeling entity as the *Control Logic Unit* (CU), and the remaining part of the controlling behaviors is called *Coordination*.

5.4.1 Control Logic Unit and Coordination

Control logic unit is a logical modeling entity to represent decision-making in manufacturing execution. We will lump the rest of the logical behaviors into coordination. While decision-making behavior can be formally described, the coordination is more difficult to be formalized. The following simple example illustrates a concrete meaning of the control logic unit and coordination.

Example: Material Dispatching

Figure 5.6 shows a representation of a station with three parallel machines and identical individual buffers. When material is loaded on LPo, we should decide where to send the material. Suppose the decision is to send the material to the machine that has minimum

WIP (Work-In-Process) among the three machines. If there is no available space in any machine, the material should be kept waiting at LPo until buffer space become available on one of the machines.

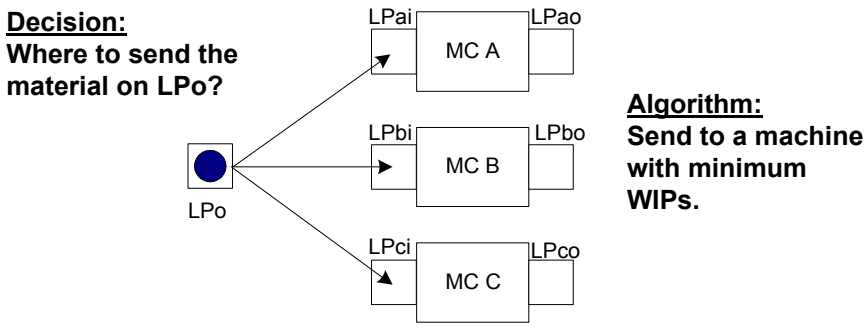


Figure 5.6 An Illustrative System for Control Logic Unit, and Coordination

In execution, there are two decision epochs for dispatching materials: i) when material arrives to LPo, and ii) when material departs from one of the machines while there exists other material blocked at LPo.

We can identify three distinctive behaviors to characterize this system: i) the decision-making behavior to find a machine with minimum WIP, ii) the physical material movement from LPo to a machine, and iii) the coordination of the decision-making behavior and physical behavior. We can capture the decision-making behavior in the form of an algorithm. The physical behavior is to execute the decision and is closely related to the physical modeling entities. The coordination behavior is to arrange these two types of behaviors properly according to trigger events such as material loading at LPo or material moving out from machines.

Therefore, controlling behaviors in manufacturing systems can be represented by three modeling entities: a *control logic unit* for decision-making behavior, a *physical modeling entity* for physical behaviors dealt in the previous section, and *coordination* for arranging interaction behavior between control logic units and physical modeling entities. In this section, we explain control logic unit, and coordination is explained in the next section.

Definition 5.5: [Control Logic Unit]

A Modeling Entity, Control Logic Unit (CU) is defined as a formal structure to represent decision-making behavior using in manufacturing execution.

5.4.2 Characterizing Decision-Making Behavior

Decision-making is a logical process to select one choice among the alternative actions that are possible. For a given situation, there are many alternative ways to make decisions, and corresponding the algorithms. As we have seen in the previous example, for dispatching material among machines, the algorithm selected a machine with minimum WIP size. However, a machine could be selected by various other ways, such as following a fixed sequence pattern, selecting a machine with minimum utilization, using scheduling algorithm to minimize the setup cost, and so on.

These decision-making behaviors in manufacturing systems are characterized by four attributes: *Name and Type*, *Internal Data Structure*, *Data Acquisition*, and *Decision Algorithm*, which are explained in detail as follows.

Name and Type

The name of a decision-making process is necessary information, because it gives a way to identify it as different from other decision-making processes. In addition, decision-making is classified with its type based on the result of the decision. In manufacturing execution, decision-making type can be characterized by such questions as *what to move*, *what to release*, *what to batch together*, and *where to move*. The answers to the questions, finally, have converged into two decision types, either *material* or *location*.

Internal Data Structure

For making a decision, we need a set of data to be used for processing a decision algorithm. The structures of data differ based on the types of decision algorithms. In the previous example, three integer variables denoted to WIP_a, WIP_b, and WIP_c, are the data required for running the dispatching algorithm.

Data Acquisition

In order to run a decision algorithm, relevant data is required in the form of Internal Data Structure. There are three different types of data depending on the sources of the data. One type is for physical entity behaviors, of which starting and end points are represented as *events*. For the previous example, when material is loaded on a machine, this event is a trigger to increase the WIP data on the machine. Hence, an event occurring due to physical behavior is a source of data to update the Internal Data Structure during manufacturing execution. The second type of data is for static information, which is not changed during manufacturing execution. For example, it includes the structural data of a

system such as distances between locations, capacities of buffers, and so on. The last type of data is triggered by the controls in manufacturing execution. For example, suppose that material at a location should be dispatched by a schedule that is created at the starting time of a shift. In this case, the schedule data should be updated periodically by a scheduling system. Such a type must be updated in the Internal Data Structure.

Therefore, once an Internal Data Structure is decided, the relevant data should be captured and updated by different methods according to the types of data. The first one is to use the initialization procedure before system execution, and another one is to use a *data-updating scheme*[5] for the dynamic events. Since detailed mappings from events to internal data structure differ from event to event, each event should have a predetermined procedure to fill or update internal data.

Decision Algorithm

A decision algorithm itself can be represented in various ways. For example, flow chart, pseudo code, or UML (Stevens and Pooley 2000) are tools to represent algorithms. However, once an internal data structure is fixed, algorithms can be represented in the form of program code too. Hence, in this thesis, we assume that a Decision Algorithm is represented using a program language with given Internal Data Structure.

5.4.3 Designing Control Logic Unit

Based on the results of characterizing decision-making behaviors in the previous section, we can design *Control Logic Unit* (CU) to describe the behavior of decision-making. In

[5] Detail discussion of Data-updating scheme is presented in section 5.7.1.

this section, we design CU by including four characterizing aspects of decision-making behaviors as follows.

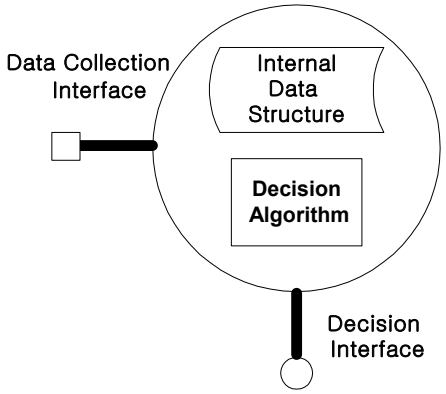


Figure 5.7 Graphical Representation of Control Logic Unit (CU)

As illustrated in Figure 5.7, CU has four elements to represent four characterizing aspects of decision-making behaviors: *Decision Interface*, *Internal Data Structure*, *Data Collection Interface*, and *Decision Algorithm*.

Decision Interface

The Decision Interface consists of the following factors representing Name and Type aspect of decision-making.

- *ControlUnitName*: Representing the name of control logic unit.
- *ControlDecisionType*: Representing the type of decision. [Location|Material]

  In manufacturing execution, there are two types of decision-making: Location Type and Material Type.
- *ControlInterface*: Representing the actual name of control interface.

- *Parameters*: Representing dynamic data used for running a decision algorithm.

Internal Data Structure

The Internal Data Structure has various forms depending on decision algorithms. However, whatever they are, the Internal Data Structure can be encapsulated as a single *Class*. If we refer to the previous example, the decision algorithm requires three integer variables to represent the number of WIP for each machine, and it can be combined as a structure, denoted by Class WIPS, containing three integer variables as follows:

Class WIPS {

Integer WIP_a, WIP_b, WIP_c;

}

Hence, Internal Data Structure can be represented as a Class containing all required data for the decision algorithm.

Data Collection Interface

The Data Collection Interface describes a mapping to convert events to elements of the Internal Data Structure. Hence, for each event, the Data Collection Interface is described by the following three elements.

- *Name of Modeling Entity*: Name of modeling entity, of which behavior occurs. The entity can be a *location* that generates the events related to physical behaviors, or it can be other *control logic units* (CUs) that generate the control events.

- *Linked Data*: Data associated with the event that has occurred. For example, when material is loaded at a location named to LP_a, we may need to get status information about the material. The format of these data is described in Linked Data.
- *Mapping Procedure*: Procedure to fill or update data in internal data structure. For example, if the internal data structure keeps WIP by the type of the material, then the corresponding mapping procedure is:

```
{
     If (_type == "A")
          WIP_a++;
     Else if (_type == "B")
          WIP_b++;
     Else (_type == "C")
          WIP_c++;
}
```

Unlike other factors (Decision Interface, Internal Data Structure, and Decision Algorithm), Data Collection Interface defined in a `CU` is only a characterizing factor interacting with other modeling entities outside the `CU`. Hence, if a `CU` is applied to different environments, Data Collection Interface is the only factor to be changed for adapting to new environments. In other words, `CU`s are highly reusable in various environments by customizing only Data Collection Interface, because all changeable parts in the `CU` are gathered in the Data Collection Interface.

Decision Algorithm

The Decision Algorithm is a coded program with Internal Data Structure. In another word, it is the detailed implementation of *ControlInterface* defined in Decision Interface.

Based on the proposed design of a logical modeling entity, Control Logic Unit (CU), the abstract class of CU is represented by OOM as shown in Figure 5.8.

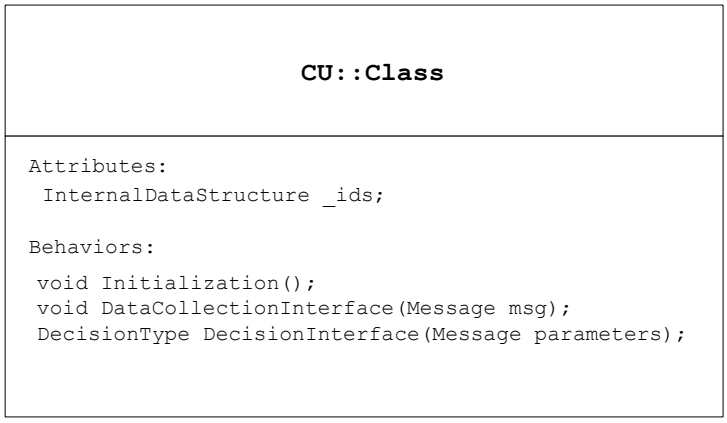


Figure 5.8 Object Model for a Modeling Entity, Control Logic Unit (CU)

5.5 Analysis of Interactions between Physical and Logical Entities

In the previous sections, we analyzed entities in manufacturing systems, and derived physical and logical modeling entities. However, these behaviors of modeling entities are not sufficient to describe the integrated behavior of manufacturing. In order to describe integrated behavior of manufacturing, the interactions among modeling entities should be formalized, which is called *Coordination*.

Figure 5.9 illustrates the meaning of coordination. As we explained, an individual modeling entity is described by its attributes and behaviors. The role of coordination is to arrange the interactions among the fundamental behaviors of modeling entities. In practice, such roles are performed by *operation workers* in manual manufacturing systems or by manufacturing execution systems (MES) in automated manufacturing systems.

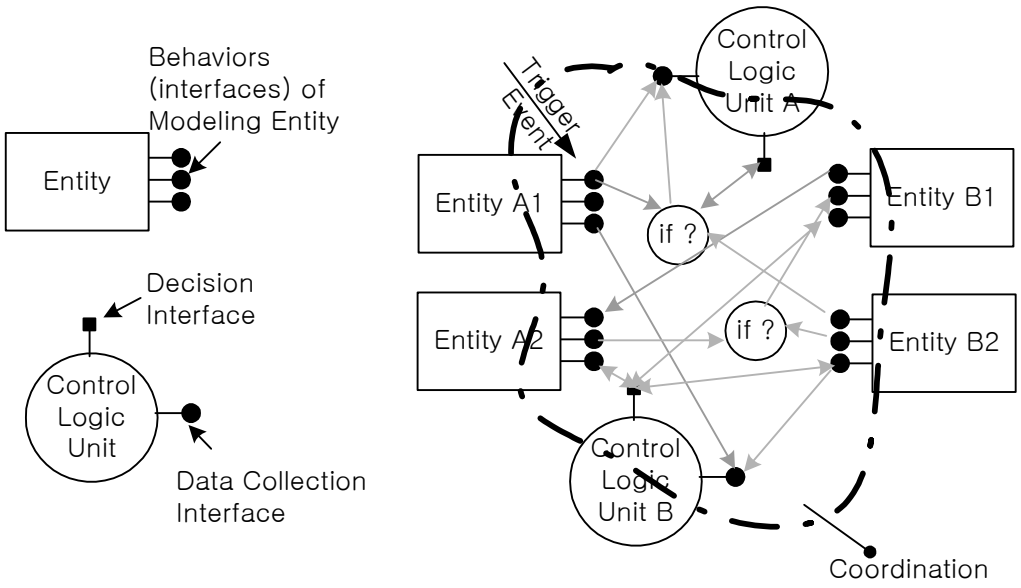


Figure 5.9 Illustration of Coordination

5.5.1 Characterizing Coordination

Coordination is a general concept in organizational theory and computer science, which is not only used in manufacturing systems, but in other fields as well. In the field of computer science, it is defined as "the process of building programs by gluing together active pieces" (Carriero and Gelernter 1992), or generally, it is also defined as "management dependencies between activities" (Malone and Crowston 1994). In this thesis, we defined coordination as a modeling entity handling interactions among the behavior of physical and logical entities.

Coordination can be represented as a structure shown in Figure 5.10, which has the following objects and behaviors.

- *Coordinator* (`CT`): A main object to handle coordinating behaviors.
- *Component* (`CP`): An object that interacts with others by the coordinator, `CT`.

- *Component Behaviors* or *Interfaces* (IF of CP): Finite number of fundamental behaviors to characterize the component driven by nature.
- *Trigger Event* (TE): An event to trigger a corresponding behavior of coordination.
- (*Trigger*) *Event Handler* (EH): A procedure of coordinating behavior for a trigger event.

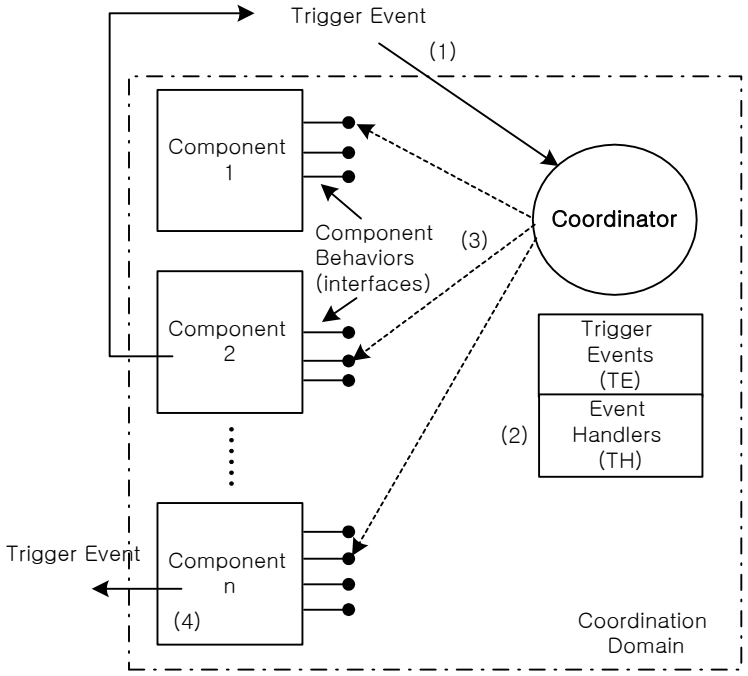


Figure 5.10 Structure and Execution of Coordination

Based on the above descriptions, *Coordination* can be defined as follows:

Definition 5.6: [Coordination]

A Modeling Entity, Coordination (CD), is formally defined as a structure of a coordinator (CT), components (CPs) and trigger events (TEs), denoted by CD = <CT, SCP, STE>, where SCP = {CP_1, CP_2,…, CP_n} and STE = {TE_1, TE_2, …, TE_m}. In here, the scope of CT, i.e. SCP, is called *Coordination*

Domain. Each CP has the component behaviors (or interfaces) that can be accessed by CT, which is denoted by $SIF_i = \{IF_i^1, IF_i^2, \ldots, IF_i^{Ii}\}$, where IF_i^j is jth component behavior for CP_i, and each TE has a corresponding event handler denoted by EH, which is defined to $\Pi_k(IF_k, fcb)$ for EH_k, where $IF_k \subset SIF = \cup SIF_i$, and FCB is a fundamental coordination behavior defined in FCB[6].

Execution Behavior of Coordination

As shown in Figure 5.10, the execution of coordination consists of the following sequence:

(1) Transmitting a trigger event to the coordinator,

(2) Choosing the event handler corresponding to the trigger event,

(3) Performing the event handling procedure by requesting the behaviors (interfaces) of components in the coordination,

(4) Updating the states of component, and generating trigger events for the same or other coordination domains, while executing component behaviors.

5.5.2 Types of Events and Component Interfaces in Coordination

One of the key attributes to characterize coordination is the set of *trigger events* and its *event handler* for each trigger event. Trigger event is an event coming from outside the coordination to which the coordinator must react. An event handler is a sequence of the

[6] FCB is a set of fundamental coordination behaviors, which are basic operations for coordination. For example, operations such as assigning a value to a certain location, comparing values, or looping control (e.g. "for-next", "do-while") are included in FCB

procedure mixed with behavior of physical and logical modeling entities (or components) in the coordination.

In coordination, anything to change the states of components is an *event*. Let us call the collection of all events to *Whole Event Set* (**E**). Among the events in **E**, any event selected for coordination behavior is called *Coordination Event* (**CE**). A sequence of **CE** is the procedure of event handling, and the **CE**s are not driven by nature but by design. In this sequence, the first **CE** comes from outside the coordination, which is called *Trigger Event* (**TE**).

A **CE** in a coordination domain can be a **TE** in other coordination domains. For example, in Figure 5.11, when material is loaded at a location of a machine (Machine `CD`) after being transported in workcenter (Workcenter `CD`), an event to load material on the machine is a **CE** in workcenter, but at the same time it becomes a trigger event in machine coordination. Using this chaining reaction mechanism with trigger events, all coordination domains are interacting to perform the integrated manufacturing behaviors in a system.

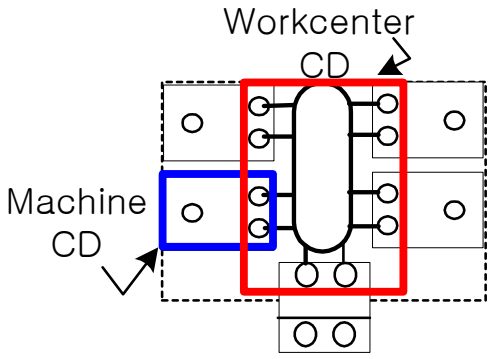


Figure 5.11 Machine Coordination and Workcenter Coordination

5.6 Formalism for Modeling Entities of Reference Model for Manufacturing (RMM)

In this section, we will present a formal structure for modeling manufacturing systems using the modeling entities identified in prior sections. The structure consists of *objects* that modelers can link to realistic components in the system and the *framework* to integrate these objects. First, we provide the notations for this formalism.

5.6.1 Notation

The following notation is used for describing RMM in formal way.

`Structure: <>`

List of elements consisting of a structure

E.g. `ST = <A, B, C>` means that structure `ST` consists of `A`, `B`, and `C` sub-structures

`Instance Of Structure: :`

E.g. `ST:st` means an instance `st` which has `ST` structure

`Element: {}`

Set of instances for a Structure

E.g. `ST` = {st_1, st_2, st_3} means a list of `ST` instances, st_1, st_2, and st_3 which has same structure of `ST`

`Sub Structure: .`

E.g. If `ST = <A, B, C>`, then `A` can be represented as `ST.A`, and sequentially, if `A = <I, J, K>`, then `I` can be represented as `ST.A.I`

`Selection: [a|b]`

List of options of which only one can be selected

E.g. ST.A:aa = [a_1|a_2|a_3] means the value of aa can bew one of a_1, a_2, or a_3.

Usage Examples:

- Structure ST = <A, B, C>, Sub Structure ST.A = <I, J, K>, and Sub Structure ST.B = <X, Y>. It represents structure ST consists of A, B, and C sub structures, and sub structure A consists of I, J, and K sub structures, and sub structure B has X and Y sub structures.
- Instance ST:st.A = <ii, jj, kk> means st, instance of ST structure, has <ii, jj, kk> structure for sub structure of A. Notational point of view, it is equivalent to ST:st.A.I = {ii}, ST:st.A.J = {jj}, and ST:st.A.K = {kk}
- ST:st.C = {c_1, c_2, c_3}, for st, it has c_1, c_2, or c_3 element as ST.C instance

5.6.2 Reference Model Formalism

ERM (Equipment Reference Model)

Equipment Reference Model (ERM) is a formal framework to describe equipment. In RMM, it describes the physical structure of manufacturing systems with Equipment Structure Model (ESM). ESM describes the structure of equipment by focusing on locations where materials can stay in the equipment. Another interesting attribute to describe the structure of equipment is the transport systems within equipment. Since there are restrictions mechanically and operationally, the possible movement from one location

to another is restricted by the transport systems in equipment. This is another factor to characterize the structure of equipment. Hence, the structure of equipment can be described as a set of locations and transport systems.

It requires quite a bit of information to describe transport systems. Hence, the detailed description of the transport system inside the equipment is separated and described in another structure called *Transportation Reference Model*, which will be explained in the next section. In ESM, only the names of transport systems are specified for linking with the detail descriptions of transport systems.

For an equipment, Equipment Structure Model (ESM) is defined as following formal structure:

ESM = <**LP, PP, BF, TS**>

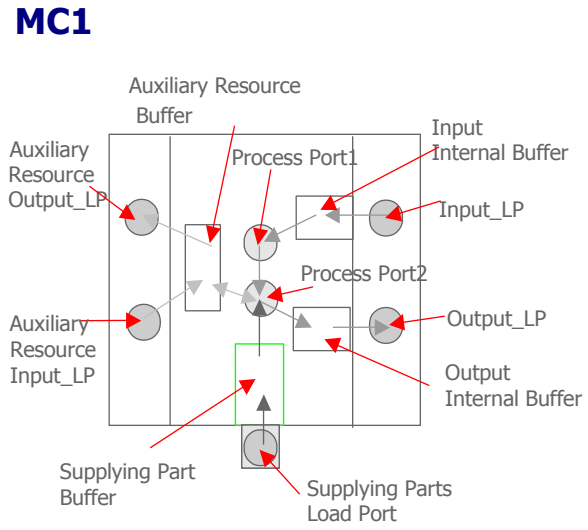
Where, **LP** = Names of load ports, **PP** = Names of process ports, **BF** = Names of internal buffers, and **TS** = Names of transport system.

ESM is a formalism to describe not only a machine but also a group of equipment. Since ESM is defined as a set of locations and transport systems, we can describe any sub-system in the factory. For an example, a workcell consisting of multiple machines and transport systems connecting the machines can be represented by ESM. The workcell is not a single machine, but because it can be represented by ESM, we can describe its structure as if it is a machine.

Examples

Process tool (MC1)

Figure 5.12 shows both graphical and ESM descriptions for a process tool named MC1. In MC1, there are three different types of materials flows. The main parts are loaded on Input_LP, and moved to Input Internal Buffer, Process Port1, Process Port2, Output Internal buffer, and Output_LP sequentially by the main part transport system. For supplying parts, after they are loaded on the Supplying Part Load Port, the parts are moved by Supplying Part TS to the Supplying Part Buffer for the consumption in processing main part. MC1 also requires an auxiliary resource for processing. Hence, the auxiliary resource modeled as *material* is loaded on Auxiliary Resource Input_LP and it is moved to the Auxiliary Resource Buffer. After being used, the material is moved to the Shared Resource Output_LP by the Auxiliary Resource TS. This verbal description about MC1 or graphical descriptions can be modeled by ESM formally.

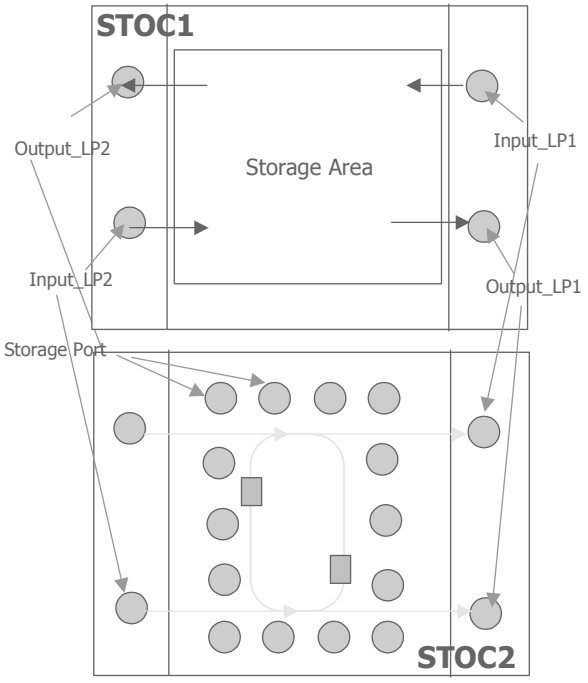



```
ESM:MC1 = <LP, PP, BF, IT>

ESM:MC1.LP = {
       Input_LP, Output_LP,
       Auxiliary Resource Input_LP,
       Auxiliary Resource Output_LP,
       Supplying Part Load Port }

ESM:MC1.PP = {
       Process Port1,
       Process Port2 }

ESM:MC1.BF = {
       Input Internal Buffer [FIFO],
       Output Internal Buffer [FIFO],
       Auxiliary Resource Buffer FIFO],
       Supplying Part Buffer [LIFO] }

ESM:MC1.TS = {
       Main Part TS,
       Auxiliary Resource TS,
       Supplying Part TS }
```

Figure 5.12 ESM model for a Process Tool, MC1

Stocker (STOC1, STOC2)

As in modeling a process tool MC1, a storage facility called *stocker* can be described by ESM. In Figure 5.13, there are two models for a stocker with different level of details. While storage area is modeled as a BF denoted by Storage_Area in STOC1, the operation policy of which is simplified with pre-defined rules, it is modeled in STOC2 as a set of 14 LPs, denoted by Storage_Ports, for describing the more detailed structure of the stocker.




```
1. ESM:STOC1 = <LP, PP, BF, TS>

ESM:STOC1.LP = {
       Input_LP1, Input_LP2,
       Output_LP1, Output_LP2 }

ESM:STOC1.PP = Null

ESM:STOC1.BF = {
       Storage_Area [FIFO] }

ESM:STOC1.TS = {
       Stacker_Crane1}

2. ESM:STOC2 = <LP, PP, BF, TS>

ESM:STOC2.LP = {
       Input_LP1, Input_LP2,
       Output_LP1, Output_LP2,
       Storage_Ports
       (SP1, SP2, ..., SP14) }

ESM:STOC2.PP = Null

ESM:STOC2.BF = Null

ESM:STOC2.TS = {
       Stacker_Crane2 }
```

Figure 5.13 ESM models for a Stocker, STOC1 and STOC2

Transporter (AGV1)

Figure 5.14 shows ESM model for a transporter. In this model, interesting locations are LoadPort and Storage_Buffer. A transport system, AGV_Internal_TS uses Robot as a transporter to transfer materials between two locations.

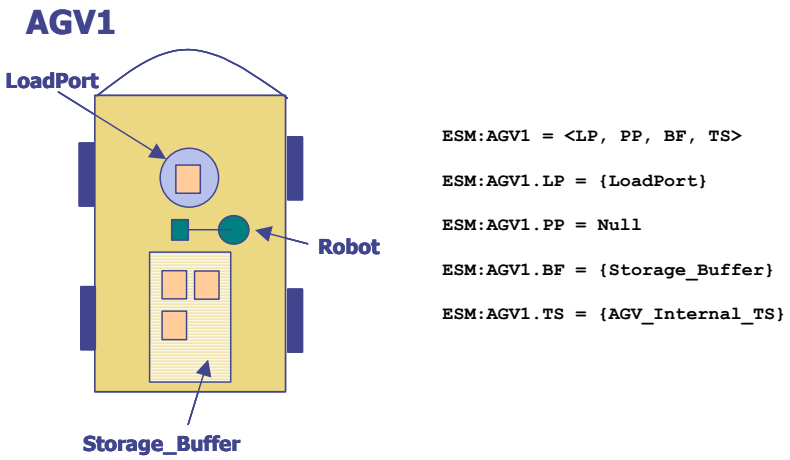


Figure 5.14 ESM model for a transporter, AGV1

TRM (Transport Reference Model)

Transport Reference Model (TRM) is another formal framework to describe the transport systems defined in ESMs. To describe a transport system, TRM uses four different modeling elements such as *Transport Structure Model* (TSM), *Transporter Specification* (TP), *Node Specification* (NODE), and *ARC Specification* (ARC), which are based on the results of analysis in previous sections.

For a transport system, Transport Structure Model (TSM) and other formations in TRM are defined as following formal structure.

TSM = <**TYPE, TP, TN, TC**>

where **TYPE** = [TG|TF|CS|RT|UT],

TP =

<TYPE,Speed,Length,AccFactor,DecFactor,WeightFactor,

InitPosition,CType,CellSize,Capacity>,

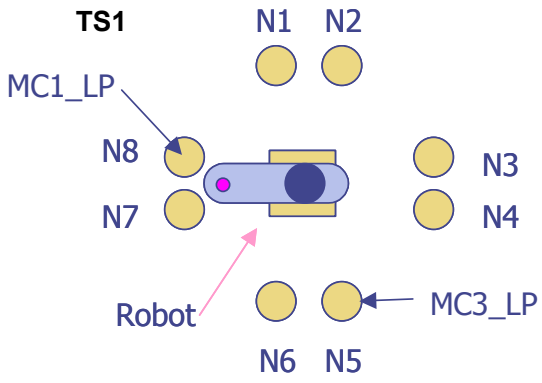
TN = <NODES, ARCS>,

where NODES = Names of nodes, ARCS = Names of arcs

Node = **<**LName**>,** where LName is the name of corresponding location linked with a node.

ARC = <SNode, DNode, Length>, where SNode is the source node name of an arc, DNode is the destination node name of an arc, and Length is length of an arc.

TC is the name of transport controller.




```
TSM:TS1=<TYPE, TP, TN, TC>
TSM:TS1.TYPE = TF
TSM:TS1.TP = {Robot}
TSM:TS1.TN = {Robot_TN}
TSM:TS1.TC = {TS1_Ctrl}

TP:Robot =
{TF,30,null,null,null,null,N1,null,1}

NODE:N1 = {null}, NODE:N2 = {null},
NODE:N3 = {null}, NODE:N4 = {null},
NODE:N5 = {MC3_LP}, NODE:N6 = {null},
NODE:N7 = {null}, NODE:N8 = {MC3_LP}

ARC:A12={N1,N2,10}, ARC:A13={N1,N3,30}
...                 ARC:A87={N8,N7,10}
```

Figure 5.15 TRM model for a Transport System, TS1

Example

Transport System (TS1)

Figure 5.15 shows both graphical and TRM descriptions for a transport system named TS1. TS1 has Transport with Free Path (TF) type, and there is a robot as a transporter, and it is linked with 8 nodes. Among these nodes, N5 and N8 are linked with physical

locations, MC1_LP and MC3_LP, respectively, and the controller of this transport system is TS1_Ctrl. All of the sufficient descriptions for the structure of the transport system are described by TRM formalism.

CRM (Control Reference Model)

Control Reference Model (CRM) is a formal framework to describe control logic units used in manufacturing execution. Based on the results of analysis and design for control logic units, CRM provides a main structure, Control Unit Model (CUM), and four sub structures such as Decision Interface (DI), Internal Data Structure (IDS), Data Collection Interface (DCI), and Decision Algorithm (DA).

For a control logic unit (CU), Control Logic Unit (CUM) and other formations in CRM are defined as the following formal structure:

CUM = <**DI, IDS, DCI, DA**>,

where,

DI = <CUName, CUType, CUInterface, Parameters>,

CUName is the name of control logic unit, CUType is the type of control decision and CUType = [LOCATION|MATERIAL], CUInterface is the name of control interface, and Parameters is required parameters for control interface.

IDS = <IDSName>,

IDSName is the name of external class[7] for internal data structure.

DCI = <MEName, LD, MPName>,

MEName is the name of the modeling entity related to data collection, LD is the component of internal data structure, which is linked to data collection, and MPName is the name of the external procedure for mapping data.

DA = <DAName>, DAName is the name of the decision algorithm.

Example

Material Dispatching (MD1)

As an example for CRM, we use the previous Material Dispatching example described in section 5.4.1 with Figure 5.6. The control logic unit explained is formally represented by CRM as in Figure 5.16.

```
CUM:MD1 = <DI, IDS, DCI, DA>
CUM:MD1.DI
   = <MaterialDispatch, LOCATION, MDispatch(), null>
CUM:MD1.IDS = {IDS_MD1} //linked with external class
CUM:MD1.DCI
   = {dci_ai, dci_ao, dci_bi, dci_bo, dci_ci, dci_co},
   DCI:dci_ki = <LPki, WIPk, WIPk ++>,
   DCI:dci_ko = <LPko, WIPk, WIPk --> k = a, b, or c
CUM:MD1.DA = { da_MD1 } //linked with external Proc.
```

Figure 5.16 CRM model for a Control Logic Unit, MD1

[7] The detailed structures of Internal Data Structure or data mapping procedure are not imbedded directly in IDS. Instead, RMM has its name to link with classes or procedures in which detail descriptions are defined.

IRM (Integration Reference Model)

Integrated Reference Model (IRM) is a formal framework to describe the coordination behavior in manufacturing execution by integrating all other modeling frameworks. Hence, Behavior Coordination Model (BCM) in IRM consists of instances of ESM, TSM, CUM, and TE (Trigger Event) for model execution. Since ESM is the unit of integration, BCM is defined based on an instance of ESM. The set of physical and logical components related to the instance of ESM forms a coordination domain.

Behavior Coordinator Model (BCM) and other sub-structures are defined as following formal structures.

BCM = <**ESM, TSM, CUM, TE**>,

where, **ESM** = Name of ESM instance of the coordination, **TSM** = Names of TSM instances defined in the coordination domain, **CUM** = Names of CUM instances required for the coordination domain, and **TE** = <EN, EH>, where TE.EN = Name of trigger event, and TE.EH = Event handling procedure.

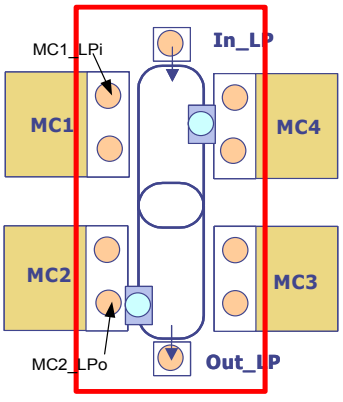


Figure 5.17 A workcell, WC1, for IRM modeling

Example

Workcenter (WC1)

As an example for IRM modeling, we consider a workcell WC1, illustrated in Figure 5.17. Though it is not a single machine, WC1 is a system of equipment, which is described by `ESM` formation. In this workcell, there are four process machines, an input loading equipment, an output unloading equipment, and a transport system with two transporters. For each processing machine, there are two load ports, one is for loading, and the other is for unloading. These load ports are denoted to `LPi` and `LPo` respectively with machine names. Since `BCM` in IRM consists of instances of `ESM`, `TSM`, `CUM`, and `TE`, we describe the `BCM` model for WS1 as following structures.

BCM:WC1 Model

BCM model for WC1 is described as follows:

```
BCM = <ESM, TSM, CUM, TE>,
BCM:WC1.ESM = {WM1}, BCM:WC1.TSM = {WT1},
BCM:WC1.CUM = {CU1}, BCM:WC1.TE = {TE0, TE1,…, TE4}
```

ESM:WM1 Model

ESM model for WC1, i.e. WM1, is described as follows;

```
ESM:WM1 =<LP, PP, BF, TS>
      WM1.LP = {MC1_LPi, MC1_Lpo, MC2_LPi, MC2_Lpo, MC3_LPi,
                MC3_Lpo, MC4_LPi, MC4_Lpo, IN_LP, OUT_LP},
      WM1.PP = WC1.BF = null,
      WM1.TS = {WT1}
```

TSM:WT1 Model

TSM model for WC1, i.e. WT1, is described as follows;

```
TSM:WT1 = <TYPE, TP, TN, TC>
      WT1.TYPE = TG, WT1.TP = {AGV1, AGV2},
      WT1.TP:AGV1 = <TG, 30, 3, 0, 0, 0, WN1, null, null, null}
      WT1.TP:AGV2 = <TG, 30, 3, 0, 0, 0, WN2, null, null, null}
      WT1.TN = {WC1_TN},
      WC1_TN.NODE = {WN1, WN2, …, WN10, IN1, IN2},
      WC1_TN.ARC = {WA12, WA23, WA34, … , IA12, IA21},
      NODE:WN1 = <MC1_Lpi>, … NODE:WN10 = <OUT_LP>
      ARC:WA12 = <WN1, WN2, 10>, … ARC:IA12 = <IN1, IN2, 5>,
      ARC:IA21 = <IN2, IN1, 5>,
      WT1.TC = {WC1_TS_Ctrl}
```

CUM:CU1 Model

CUM model for WC1, i.e. CU1, is described as follows;

```
CUM.CU1 = <DI, IDS, DCI, DA>,
      CU1.DI = <Dispatch1, LOCATION, Dispatch1(),
                                        "Location _srcLoc">,
      CU1.IDS = {IDS_CU1}, //IDS_CU1 is data structure for fixed routing
      CU1.DCI = null,
      CU1.DA = { da_Dispatch }
```

TE:TE0, TE1, …, TE4 Model

`TE:TEi = <EN, EH>`, `i` = 0, 1, …, 4, denoting machine index. 0 is `IN_LP`.

`TEi = <LP_i.LoadMaterial, EventHandler_i>`, where `LP_i` is the input load port of machine `i`.

For example, when a trigger event, `IN_LP.LoadMaterial()`, i.e. material is loaded at `IN_LP`, occurs, the corresponding event handler, `EventHandler_0`, is described as follows:

```
EventHandler_0
{       Location _Dest = WC1.CUM:CU1(IN_LP);
        WC1_TS_Ctrl.MoveOrder(_Dest);
        WaitUntil[8](ReadyToLoad());
        AdvancedTime[9] (UnloadTime);
        WC1_TS_Ctrl.LoadMaterial(IN_LP.UnloadMaterial());
        WaitUntil(ReadyToUnload());
        _Dest.LoadMaterial(WC1_TS_Ctrl.UnloadMaterial());
}
```

5.6.3 Database Representation of RMM

Since RMM is represented with formal structures, RMM can be converted into a database easily. In this section, we present a database representation of RMM.

RMM mainly consists of four different structures such as `ESM`, `TSM`, `CUM`, and `BCM`. Each structure also consists of sub-structures and/or instances of structures. Since each structure can be converted to a *table* in database, and a sub-structure should have a link to the high level structure, the RMM formalism can be converted to the forms of database tables mechanically. Figure 5.18 shows a way of database conversion, and Table 5.2

[8] WaitUntil (*cond*) is a description primitive, which means to wait until the condition, *cond*, becomes true
[9] AdvancedTime(*time*) is a description primitives, which means to wait *time* units

illustrates the detailed structure of database tables for RMM. The relationships among tables are represented in Figure 5.19.

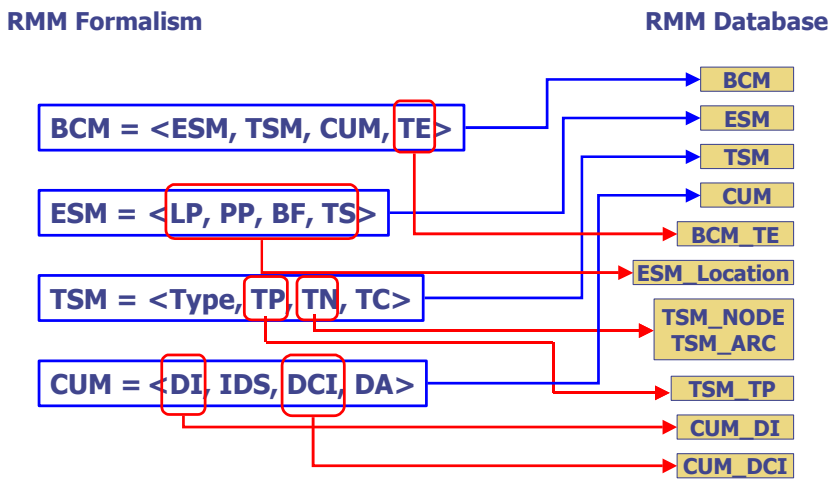


Figure 5.18 Database Conversion for RMM

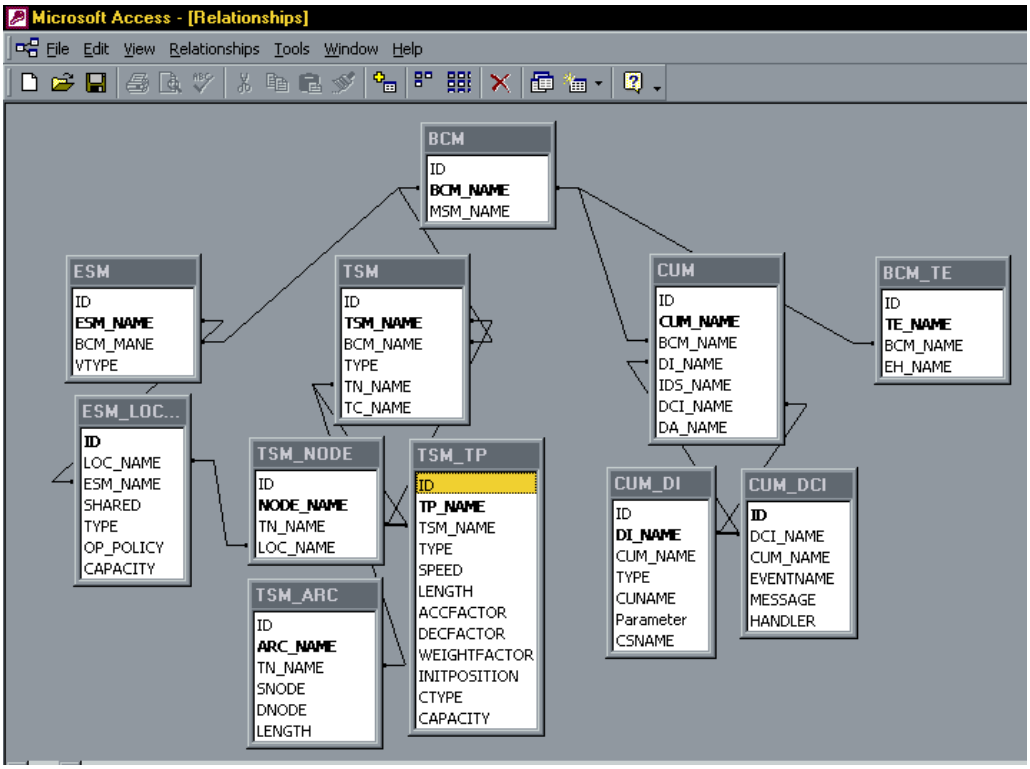


Figure 5.19 Relationships of Database Tables in RMM

Table 5.2 Database Tables for RMM

Database Table Name	Field Name	Descriptions
BCM	BCM_NAME	Name of behavior coordinator
	MSM_NAME	Name of Manufacturing System Model. The set of BCs with same MSM name is configuring a factory model
ESM	ESM_NAME	Name of equipment model
	BCM_NAME	Name of BC related with the equipment model
	VTYPE	Flag checking if the equipment is virtual machine or not
TSM	TSM_NAME	Name of transport system model
	BCM_NAME	Name of BC related with the transport system model
	TYPE	Type of transport [TG|TF|CS|RT|UT]
	TN_NAME	Name of transport network for the TSM model
	TC_NAME	Name of external transport controller related to TSM model
CUM	CUM_NAME	Name of control logic unit model
	BCM_NAME	Name of BC related with the control logic unit model
	DI_NAME	Name of decision interface related with the control logic unit model
	IDS_NAME	Name of external class for internal data structure
	DCI_NAME	Name of data collection interface related with the control logic unit model
	DA_NAME	Name of decision algorithm defined externally
BCM_TE	TE_NAME	Name of trigger event model
	BCM_NAME	Name of BC related with the trigger event model
	EN_NAME	Name of trigger event
	EH_NAME	Name of event handler for the trigger event
ESM_LOCATION	LOC_NAME	Name of Location
	ESM_NAME	Name of equipment mode in which the location is defined
	SHARED	Flag to check if the location is a shared one or not
	TYPE	Type of location [LP|PP|BF|TS]
	OP_POLICY	Operation policy for Buffer (BF) type location
	CAPACITY	Capacity of Buffer (BF) type location
TSM_NODE	NODE_NAME	Name of transport node
	TN_NAME	Name of transport network including the node
	LOC_NAME	Name of location related to the transport node
TSM_ARC	ARC_NAME	Name of transport arc
	TN_NAME	Name of transport network including the arc
	SNODE	Name of source node
	DNODE	Name of destination node
	LENGTH	Length of the arc, between source and destination
TSM_TP	TP_NAME	Name of transporter model
	TSM_NAME	Name of TSM model related with the transporter model
	TYPE, SPEED, …, CAPA	Fields for describing characteristics of transporter
TSM_TYPE	ID	Id for TSM Type
	TYPE	Type of Transport System, [TG|TF|CS|RT|UT]
CUM_DI	DI_NAME	Name of the decision interface
	CUM_NAME	Name of CUM model related to the decision interface
	TYPE	Type of decision interface, [Material|Location]
	CUNAME	Name of control unit
	PARAMETER	Parameters used for control logic
	CSNAME	Name of control system in which the control logic unit is implemented
CUM_DCI	DCI_NAME	Name of data collection interface
	CUM_NAME	Name of CUM model related to the data collection interface
	EVENTNAME	Name of behavior event related to data collection
	MESSAGE	Message format transmitted to control logic unit
	HANDLER	Message handling procedure

5.7 Modeling and Implementation Issues in RMM

Although RMM provides a framework for describing discrete-part manufacturing systems, there are some modeling issues that are dependent on the modeling options in system specification. In this section, we discuss the following three issues: i) *Internal Data Update Scheme*, ii) *Control System Architecture*, and iii) *Specifying Coordination Domain*.

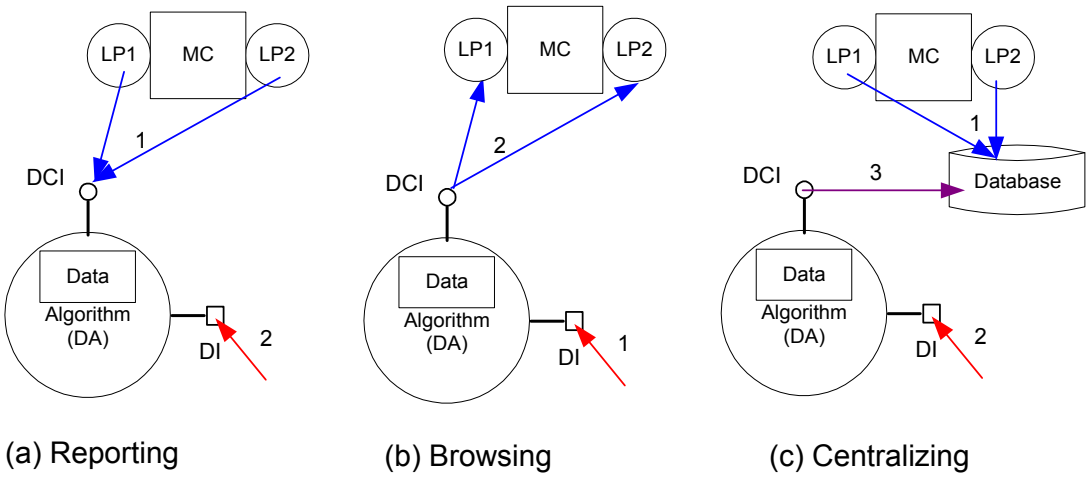


Figure 5.20 Three Internal Data Updating Schemes

5.7.1 Internal Data Updating Schemes

In RMM, control logic units (CU's) were represented with 5 elements, which are Decision Interface (DI), Internal Data Structure (IDS), Data Collection Interfaces (DCI), and Decision Algorithm (DA). Whenever it receives a decision request from a coordinator through DI, a CU executes the DA with internal data stored in IDS, and returns the control decision to the coordinator. In order to exhibit this behavior of the control logic unit, the data on the IDS should always be the updated data representing the current

factory floor status correctly. The method to maintain the data in IDS with the latest one is called *Data Updating Scheme*.

For updating internal data in control logic units, there are three schemes such as i) *Reporting*, ii) *Browsing*, and iii) *Centralizing*, as shown in the Figure 5.20.

Reporting Method

In the Reporting Method, when an event (or behavior) defined in DCI occurs, the event is reported to CUs directly and is used for updating the internal data. Later, since the internal data is always maintained with the latest data through this method, when a coordinator requests a decision making, CU can execute its decision algorithm (DA) directly without gathering any data.

Browsing Method

Unlike the Reporting Method, when an event defined in DCI occurs, the event is not reported to the CU directly in the Browsing Method. Instead, if a coordinator requests to a CU, then the CU starts to search and gather the status information from the factory floor. After finishing data gathering, the CU executes its DA and returns the result to the coordinator.

Centralizing Method

The Centralizing Method uses a central database to provide data for CUs. In this method, when an event defined in DCI occurs, the event is reported directly to the central

database, and later, if a coordinator requests to a CU for making a decision, the CU gathers data required for executing the algorithm from the central database, and after executing the DA, it returns the decision to the coordinator.

In real manufacturing execution, these three methods are used together. Some of the data can be gathered with the Reporting Method, and other data can be collected with the Browsing Method or the Centralizing Methods. Hence, in order to model these data updating behaviors precisely, modelers should designate a data updating method for each data. However, it is neither easy nor efficient for mapping data with its updating methods in the simulation environment. Fortunately, under an assumption that computing times for data process and communication can be ignored[10], these three updating methods have equivalent behaviors in the decision-making processes. Since the computing times, compared with physical processing times, are small enough to be ignored, the assumption can be justified.

Therefore, we can select one of these as a standard method for data updating scheme. To select a proper method, we need to analyze the pros and cons of each method. In the Reporting Method, a main benefit is that CU does not take any time for gathering required data, when a coordinator requests a decision, because this method always maintains the data on IDS with the latest status. However, it requires more communication traffic than

[10] Actually, these methods have different computing performance. In case of the reporting method, it can execute the decision algorithm directly after receiving a request from coordinator, but in browsing method, it can execute decision algorithm only after gathering the status data from factory floor. On the other hand, whenever events occur, the reporting or the centralizing method uses the computing resource to communicate events, but the browsing method consume less computing resource for the communication.

other methods, because a physical modeling entity of which status is updated should send the updated information to all CUs linked with the entity.

Regarding communication traffic, the Centralization Method is efficient because all messages are sent to the central database only once. However, when we use a central database, we need to consider how to design the database to cover all information requirements for decision-making in CUs. Especially, since the database schema should be robust to the changes of information requirements of CUs, the data stored in the database are to be with raw level. Therefore, a CU requires a filter to generate data from the raw data in central database, so it takes longer time to prepare a set of internal data for executing the decision algorithm (DA).

In case of the Browsing Method, it also takes times for gathering data before executing a decision algorithm as the Centralizing Method. However, the more serious problem is that this method has a functional limitation that it is difficult in using past data. A characteristic feature of the Browsing Method is to request necessary data to the physical modeling entities, when a CU receives a decision making request. However, since physical modeling entities provide only the current status information, this method is not appropriate for the decision algorithms that requires the past trajectory data.

Therefore, we selected the Reporting Method as the data updating scheme for RMM. Although it takes more time for communicating messages, it has a benefit to execute decision algorithms faster than other methods. Especially, in using the model generation

procedure explained in the next chapter, the dynamic mapping between CUs and events can be generated easily from RMM descriptions by using this method.

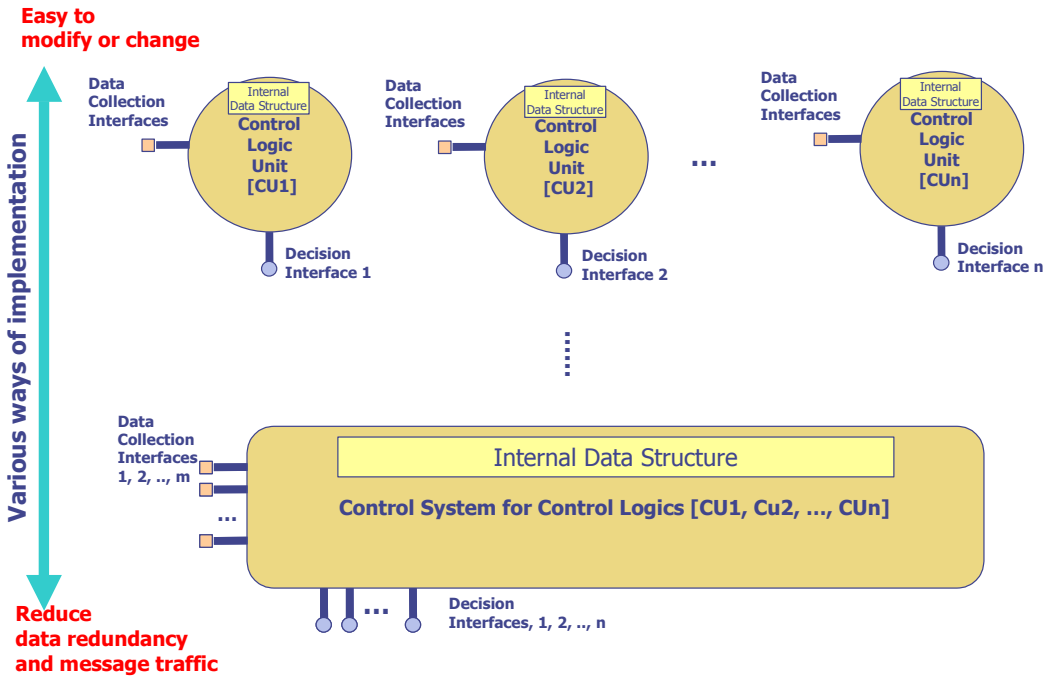


Figure 5.21 Design Alternatives of Control System Architecture

5.7.2 Control System Architecture

In RMM, Control Logic Unit (CU) is a conceptual structure to represent a decision-making behavior. These CUs can be realized with various implementation structures according to how to design the control architecture. One extreme structure is to implement each CU as a control system, and another extreme is to implement all CUs in a single control system shown as in Figure 5.21. In between two extreme structures, there are various alternative architectures for control systems.

Although any control architecture behaves equivalently because control behaviors depend on only internal data (IDS) and decision algorithms (DA), they can have different performance in simulation execution. In designing control system architecture, there are two trade-off factors: one is related to system performance, and the other is related to management of CUs. For example, when all CUs are implemented as a single control system, then we can minimize data redundancy and message traffic for updating internal data. However, when we adopt new decision algorithm in a CU, the structure of IDS for the CU is changed and it can influence other CUs that share the IDS in the same control system. On the other hand, if each CU is implemented as its own control system, then while the data redundancy and message traffics are increased, the new decision algorithms of a CU can be easily replaced without changing other CUs.

Since there can exist many structures between the above two extremes, to maximize both system performance and efficiency of management, the system modeler should decide how to arrange the CUs to control systems. In RMM, CUM is modeled as <DI, IDS, DCI, DA>. However, in order to add the flexibility in control architecture design, we can add one component, CS to designate the name of the control system in which the CU is implemented. So, by modifying CUM with <DI, IDS, DCI, DA, CS>, RMM provides a way that a modeler can specify the control system architecture with various ways.

5.7.3 Design of Coordination Domain

In RMM, equipment is defined as a set of locations with transport systems on which materials can be placed, and it is modeled in the form of ESM. For example, Figure 5.22 illustrates a machine that consists of five locations (input/output load ports, buffers and process port) and four transport devices (TS1, TS2, TS3, TS4). The ESM model (M) for the machine can be represented as ESM:M = <{In_LP, Out_LP}, {PP}, {In_BF, Out_BF}, {TS1, TS2, TS3, TS4}>.

Suppose that the machine is divided into two parts by a dotted line as in (b). Then, this machine can be modeled by three different ESMs named M1, M2, and M3 as follows. ESM:M1 = <{In_LP}, {PP}, {In_BF}, {TS1, TS2}>, ESM:M2 = <{Out_LP}, null, {Out_BF}, {TS4}>, and ESM:M3 = <null, {PP}, {Out_BF}, {TS3}>. Any location or transport system defined in ESM:M is included once in one of ESM:M1, M2, or M3, unless the location is shared in multiple ESM models.

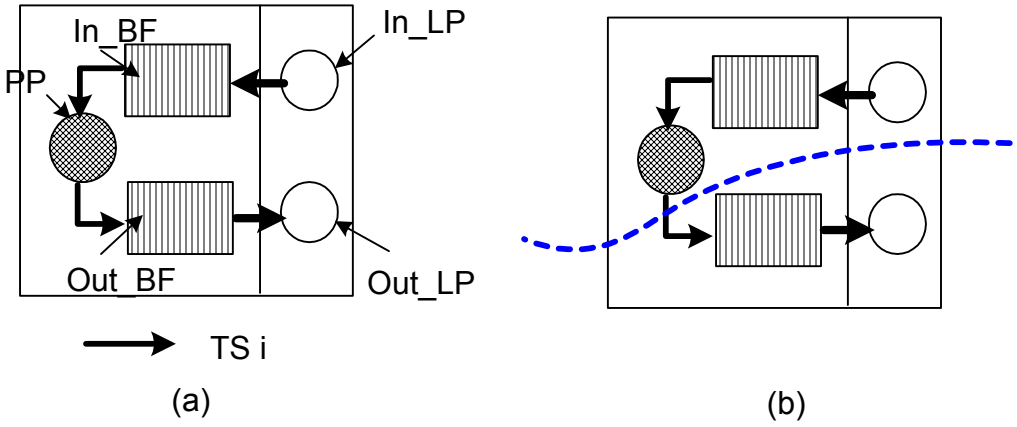


Figure 5.22 Illustrations for Separating Equipment Model

Although the machine model, ESM:M is divided into three models, they have same execution behaviors. In RMM, a unit of the coordination model is based on an equipment model (ESM). Hence, since ESM:M is separated into M1, M2, and M3, these separate models require corresponding behavior coordination models (BCM). These BCM:M1, M2, and M3 models can be derived from BCM:M without any differences in coordination behavior.

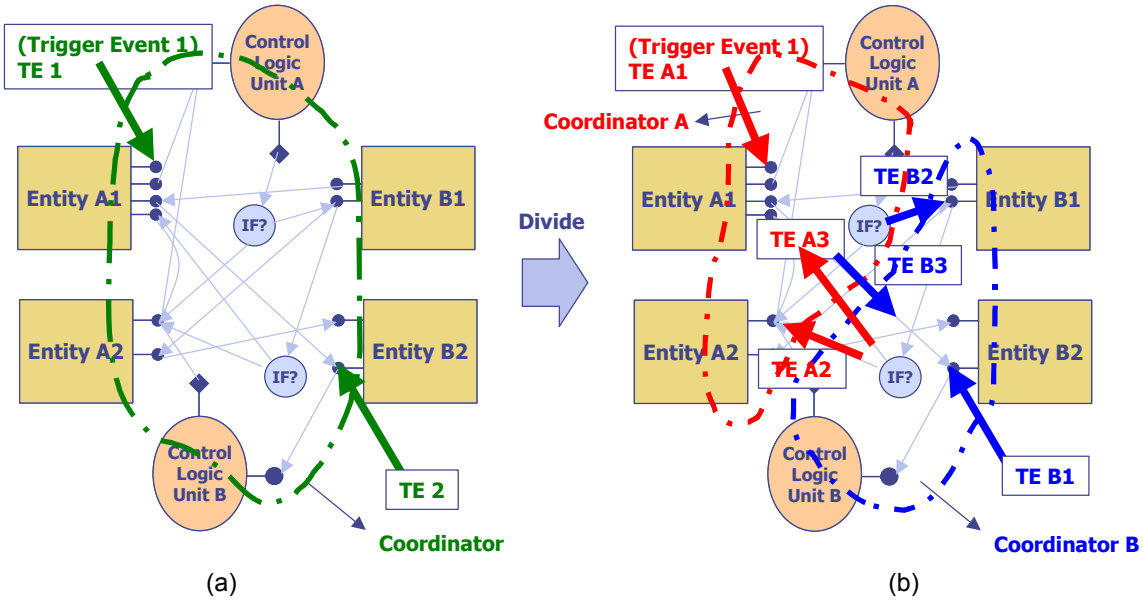


Figure 5.23 Illustrations for Separating Coordination Domain

Figure 5.23 illustrates the separation of behavior coordination. In (a), a coordination (CD) is defined with 6 components and two trigger events, and denoted to CD= <CT, {A1, A2, B1, B2, CU_A, CU_B}, {TE1, and TE2}>. The coordination, CD, can be separated with two coordinations, CD_A and CD_B, as follows. CD_A = <CT_A, {A1, A2, CU_A}, {TE1, TE_A2, TE_A3}>, and CD_B = <CT_B, {B1, B2, CU_B}, {TE2, TE_B2, TE_B3}>. Since

a coordination event in a CD_A (or CD_B) passing to other CD_B (or CD_A) is changed to a trigger event to CD_B (or CD_A), the separated coordinations behave in exactly the same sequences as in the original coordination.

Therefore, designing coordination domains does not influence the behavior of coordination. However, it has meaning in the points of modularization of system descriptions. An extreme example is to suppose that a factory is modeled with single coordination domain. Then, all of the locations and transportation systems are described with an ESM model, and trigger events and their event handling procedures should be described as a BCM model. In this case, it is be very difficult to model the big system in a single coordination domain because of modeling complexity. Hence, we should take advantage of reducing modeling complexity with modularization. However, for this, how to divide coordination domains is an issue in modeling.

One of the important criteria for designing coordination domains is dependent on how much the designed coordination domains can be reused in different environments. Since a behavior coordinator is an integrated entity behaving as an individual object, the coordinator domain should have semantic meanings for reusing it in other environments. For example, if machine is a unit for domain coordination, then a BCM model for the machine can be reused in other environments by modifying descriptions that are dependent on environment, because CUM.DCI and BCM.TE.EH are the only descriptions depending on environment.

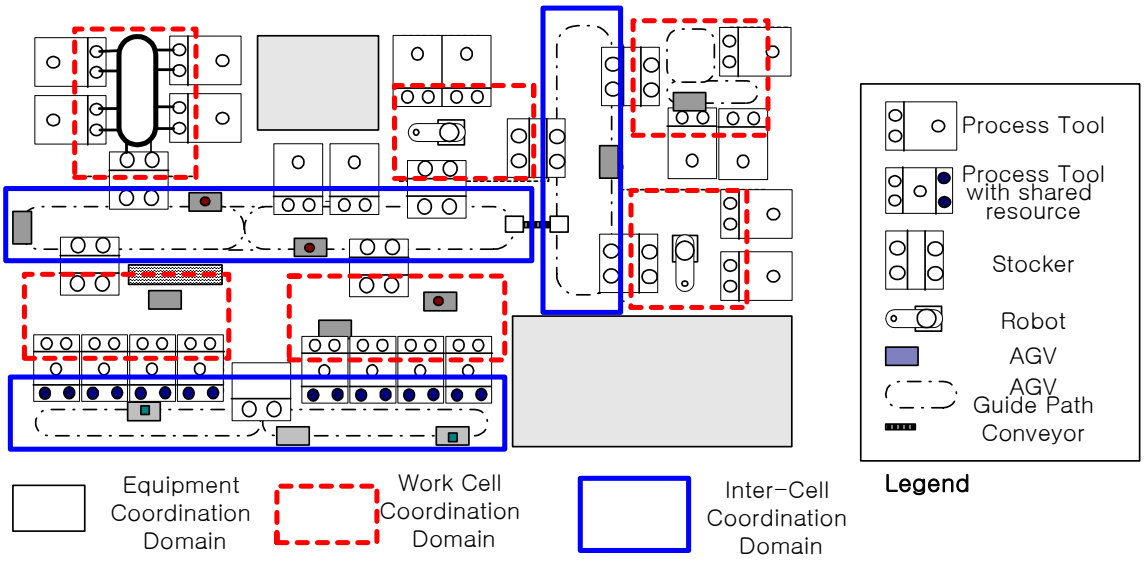


Figure 5.24 Example of Design of Coordination Domains

Therefore, in RMM, physical equipment and virtual equipment are the unit for designing coordination domains for reusing system descriptions efficiently. Figure 5.24 shows an example of specifying coordination domains with this criterion.

5.8 Summary

In this chapter, we constructed a formal framework to describe a specific domain for discrete part manufacturing systems. The domain is generic so that many modern manufacturing systems are included in this domain. First, we analyzed the domain to find *modeling entities* in which we observe and generate interesting events that can occur in manufacturing execution with a viewpoint of the discrete event system.

Based on the results of domain analysis, we found three distinctive characteristic types of modeling entities for *physical entities*, *logical entities*, and *interactions* between the two in manufacturing execution. Physical modeling entities includes *Material* and *Location*

that can also be divided into *Load Port*, *Process Port*, and *Buffer*, and *Transport System*. . Since there are distinctive types of transport systems in manufacturing systems, we characterized the factors to describe the transport system with separate common interfaces and type-specific behaviors. As a logical modeling entity, *Control Logic Unit* (`CU`) is formalized for decision-making behaviors, and in order to coordinate the physical and logical modeling entities, we formalized the behaviors of *Coordination* as a modeling entity.

Reference Model for Manufacturing (RMM) a framework to link these modeling entities derived from domain analysis. This formalism allows system descriptions to be standardized in the forms of database schema. In other words, it means any manufacturing systems in the domain can be described and modeled with the same formal structures in the forms of relational database. Once the structure of data is fixed and the data is sufficient for describing a target system, we can apply a procedure to convert the system descriptions to an executable simulation model. Since this automation procedure is independent of the types of manufacturing systems within this framework, it can generate simulation codes for any manufacturing system in the domain without modifications.

CHAPTER VI

DESIGN AND IMPLEMENTATION OF MODEL GENERATION PROCEDURE FOR RMM

In the last chapter, we identified modeling entities through domain analysis, designed a Reference Model for Manufacturing (RMM), and discussed domain knowledge governing system execution. In this chapter, we design and implement a model generation procedure for RMM. As shown in Figure 6.1, the final goal of the Reference Model Approach is to generate an executable simulation model from the RMM descriptions. In order to generate an executable simulation model, we need a simulation execution mechanism (SEM) to run a simulation model, and should define a simulation model structure that is to be executed in the execution mechanism.

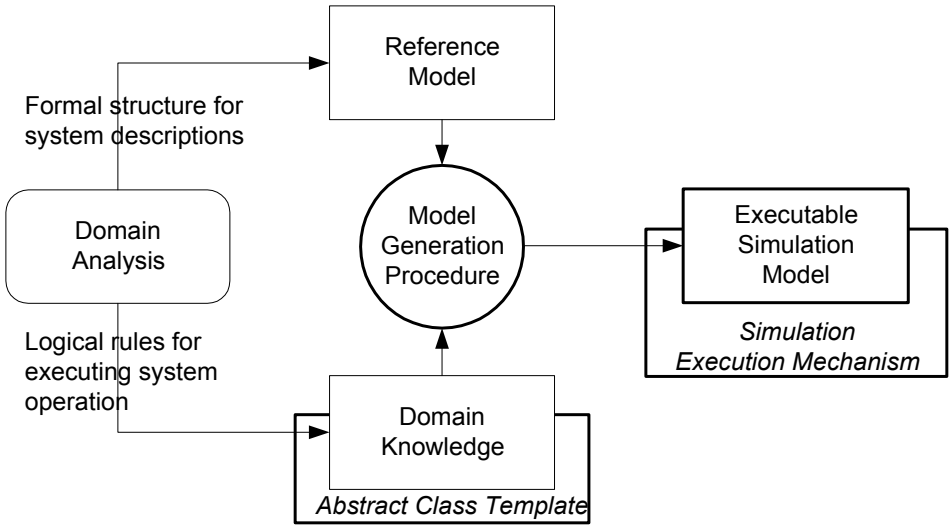


Figure 6.1 Framework for Model Generation Procedure

A simulation model for RMM is structured from three sources: i) *System descriptions in RMM format*, ii) *Domain knowledge for system operations*, and iii) *Simulation execution mechanism interacting with simulation models*. While a model designer gives the system descriptions for a target system, the other information should be provided by systematic procedures structuring domain knowledge with simulation execution mechanism. In order to facilitate the model generation, the systematic procedures should be designed and implemented to match the structure of RMM.

In this chapter, we first introduce a simulation execution mechanism (SEM) for RMM that has a structure to match RMM. Based on the proposed SEM, domain knowledge is structured in the form of *Abstract Class Templates* (ACT). Abstract Class Template is a skeleton program that can be dynamically configured according to the RMM descriptions. With the Abstract Class Templates, we can easily design a model generation procedure; once a target system is described in the form of RMM, the model generation procedure creates program *Classes* constructing an executable simulation model by configuring ACTs. Since the generated program Classes are designed to be executable under the proposed SEM, we can generate an executable simulation model by integrating these program Classes.

6.1 Simulation Execution Mechanisms

6.1.1 General Simulation Execution Mechanism for Discrete Event Simulation

To design a SEM for RMM, we need to understand the general SEM for discrete event simulation. Depending on the simulation worldviews (Fujimoto 2000), there are three

types of execution mechanisms: *Event Driven Execution* (EDE), P*rocess Oriented Execution* (POE), and *Activity Scan Execution* (ASE). The first two execution mechanisms are the more widely used among the three views. All three mechanisms are equivalent in the sense that POE and ASE can be translated into semantically equivalent EDE (Perumalla and Fujimoto 1998).

Therefore, we use EDE to explain the simulation execution mechanism for discrete event simulation in this section.

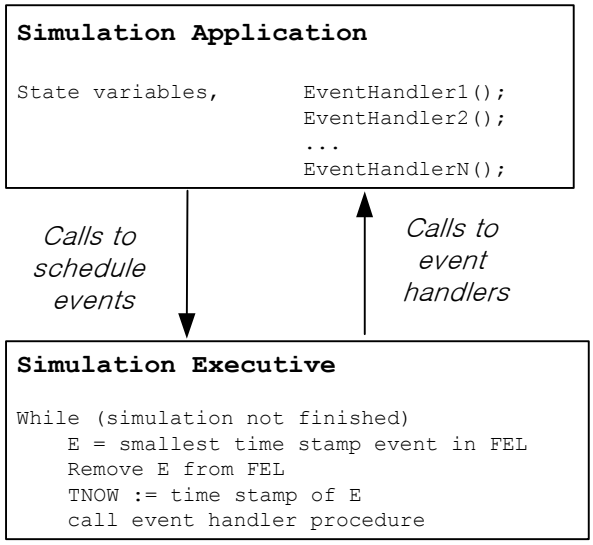


Figure 6.2 Event Driven Simulation Execution Mechanism

The discrete event simulation programs under EDE are divided into two parts as illustrated in Figure 6.2. One part is called *simulation application*, which is a simulation model for imitating the behaviors of a target system, and the other part is called *simulation executive*, which enables various simulation applications to be executed.

While simulation applications change according to target systems, simulation executive is invariant in relation to simulation applications.

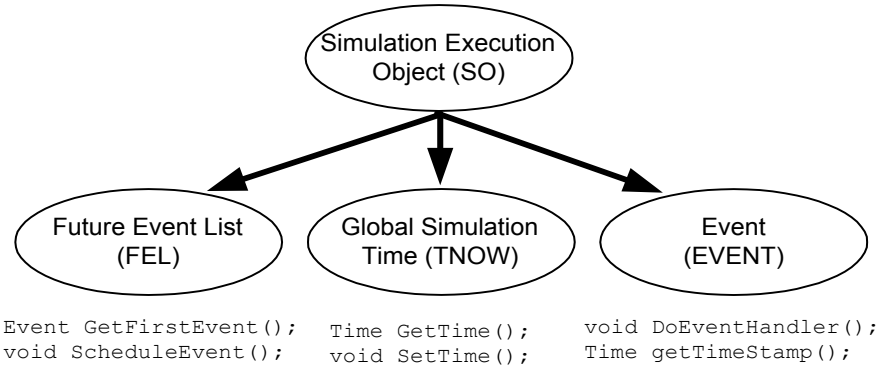


Figure 6.3 Object Model for Simulation Execution

The two main roles of simulation executive are i) *event management* and ii) *simulation time management*. To perform these roles, as Figure 6.3 shows, three special data structures, called simulation execution objects (SOs), should be defined in the simulation executive. The *Future Event List* (`FEL`) is a sorted linked list for storing future events with time stamps. The *Global Simulation Time* (denoted by `TNOW`) is a special data structure providing current simulation time information, and *Event* (`EVENT`) is a data structure for representing simulation events, which consists of event handler and event occurrence time (time stamp) information.

As Figure 6.2 illustrates, simulation execution mechanism starts from an event with the smallest time stamp in `FEL`. The current simulation time (`TNOW`) is updated to the time stamp of the event, and the simulation executive calls the corresponding event handler

modeled in simulation application. When an event handler is called, it updates state variables and schedules future events caused by the current event using `ScheduleEvent()` method in `FEL`.

6.1.2 Simulation Execution Mechanism for RMM

In this section, we propose a simulation execution mechanism for RMM. Since RMM has multiple aspects for describing manufacturing systems, the proposed SEM is designed for reflecting the structures of RMM to facilitate the executable simulation model generation.

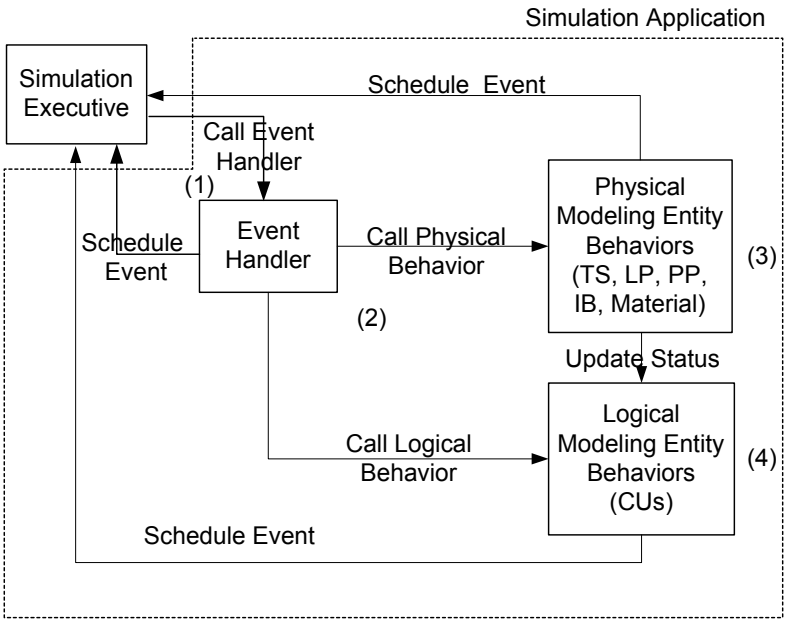


Figure 6.4 Simulation Execution Mechanism for RMM

As Figure 6.4 illustrates, the execution mechanism for RMM is based on the Event Driven Execution (EDE) mechanism that consists of two parts: simulation application and simulation executive. However, while simulation application in general EDE

mechanism consists of a set of event handlers, simulation application in the proposed execution mechanism is structured to reflect the special features of RMM.

A simulation application in the proposed execution mechanism consists of three main parts: *event handler*, *physical modeling entities*, and *logical modeling entities* as RMM structure, (i.e. `BCM = <ESM, TSM, CUM, TE>`). It is well matched with the RMM structure; i.e. the procedures of trigger event handlers are matched with the contents of `BCM.TE.TH`, the behaviors of physical modeling entities are also related to locations and transport systems defined in `ESM` or `TSM`, and the behaviors of logical modeling entities correspond to the contents of the `CUM` model. Its detail execution mechanism is described in the following sections.

Behaviors of Simulation Executive (1)

As in the general simulation executive, the role of the simulation executive for RMM is to manage events and simulation time. If there are events in `FEL`, the simulation executive selects an event with the smallest time stamp, updates the `TNOW` to the time stamp of the event, and calls the event handler modeled in simulation application. Since the execution mechanism for RMM is based on the general simulation executive, the proposed mechanism can be deployed without additional development of special simulation executive for RMM.

Behaviors of Event Handler (2)

The roles of event handlers are matched with the behavior coordinators described in the BCM.TE structure in RMM. Hence, coordination behaviors described in TE.EH for a trigger event (TE.EN) will be used to configure the contents of event handler. In RMM, the coordination behaviors can be classified with the following four types:

- *Request/Response behaviors with physical modeling entities*

  In RMM, the physical modeling entities represent the behavior of physical objects such as Locations (LP, PP, and BF), Transport Systems (TS), and Materials (Material), which define the physical structure of manufacturing systems. Each of these physical modeling entities has own behavior. For example, a load port (LP) has behaviors such as LoadMaterial(), and UnloadMaterial(). Hence, when an event handler needs to perform a physical behavior, it requests the physical modeling entity to perform the physical behavior, and gets the results back for handling the next procedure.

- *Request/Response behaviors with logical modeling entities*

  The logical modeling entities represent decision-making behaviors in the form of control logic unit (CU) defined in RMM. Each CU has a <DI, IDS, DCI, DA> structure, and if it receives a request from an event handler for decision-making, then it executes decision algorithm (DA) based on internal data (IDS), and sends in the result as a response through decision interface (DI).

- *Simulation execution related behaviors*

  While coordinating the component behaviors, an event handler may need to use such functions of simulation execution objects as ScheduleEvent(),

`AdvanceTime`(), and `WaitUntil`(). Hence, the event handler should have a reference pointer to the *simulation execution objects* (SOs) to call simulation execution behaviors.

- *Fundamental coordination behaviors* (FCB)

  FCB is a set of basic operations for gluing together other component behaviors, which are characterized by four different types: assigning, comparing, conditioning, and looping.

Behaviors of Physical Modeling Entities (3)

When an event handler requests a behavior of a physical modeling entity, three types of activities are performed: i) updating status information, ii) reporting status changes to the logical modeling entities that require the information to update their internal data, and iii) scheduling future events that trigger other event handlers. For example, when material (`mt`) is loaded at a load port (`LP1`) and the event handler calls the physical behavior (`LoadMaterial`()) of the physical modeling entity (`LP1`), then its program structure is abstracted as follows:

```
(1)   Class LP1()
(2)   {
(3)         Material _material; //Status variable
(4)         CU1 _cu1; CU2 _cu2;//CUs requiring status change update
(5)         SimulationObject _so;
            …
(6)         void LoadMaterial(Material mt)
(7)         {
(8)               _material = mt;  // i) Update status variable
(9)               _cu1.updateInformation(message);// ii) Reporting status
(10)              _cu2.updateInformation(message);//       change to CUs
(11)              ….
(12)              _so.ScheduleEvent(event);// iii) Scheduling event
(13)        }
            …
      }
```

When LoadMaterial event occurs in this example, line (8) simulates the fundamental behavior of loading material on the load port by assigning loaded material mt to internal status variable _material, and as shown in line (9, 10), this update information is reported to _cu1, and cu2 control logic units for updating the internal data structure of the control logic units. If LoadMaterial() triggers another event handler, a new future event is scheduled as in line (12).

In RMM, such a physical modeling entity is described in the form of ESM (ESM = <LP, PP, BF, TS>) for a coordination domain, and these descriptions are used to build a set of programs for the physical modeling entity. For updating status information, a physical modeling entity should know the list of control logic units that require the status change information of the entity. Since these descriptions are defined in CU.DCI (data collection interfaces) of each CU, we can get the list of CUs by browsing through DCI's of CUs in RMM model.

Behaviors of Logical Modeling Entities (4)

Similar to a physical modeling entity, a logical modeling entity has three types of behaviors: i) executing its decision algorithm and returning the decision result to the event handler requesting a decision-making of the control logic unit, ii) updating internal data, when it receives status updating message from physical modeling entities, and iii) scheduling future events that are driven by the execution of logical behaviors. The descriptions for the logical modeling entities are modeled in the form of CUM in RMM, and these are used for building up a set of programs for control logic units (CUs).

6.2 Design of Simulation Model Structure for RMM

In the previous section, we proposed a simulation execution mechanism for RMM, which consisted of three types of entities: *Event Handlers*, *Physical Modeling Entities*, and *Logical Modeling Entities*, and we also defined the characteristic behaviors of each type of entity in detail.

Based on the proposed execution mechanism, we can design the structure of simulation models in the forms of *Abstract Class Templates*. Abstract Class Template is a common programming template for entities that have same structures but different contents. For example, all load ports (`LP`s) have the same structure as an `LP` but each of them has different control logic units (`CU`s) to which load ports report its updated information. Hence, they cannot be modeled as a single class[11], but each of them ought to be modeled as its own LP class. To build these LP classes that have their own behaviors efficiently, we can capture the common structure of these LP classes as a template. It is called the Abstract Class Template for LP, or simply called the *LP Class Template*, and all classes derived from the LP Class Template are called *LP Type Classes*.

6.2.1 Abstract Class Template Model for RMM

Figure 6.5 illustrates a model of abstract class templates used for RMM. It consists of eight class templates, and three interfaces. Interface is a definition of behaviors, in which any class linked with the interface should implement the behaviors defined in the interface.

[11] In here, *class* means the concept of class defined in OOP, which is a blueprint, or prototype, that defines the attributes and the behaviors common to all objects of a certain kind.

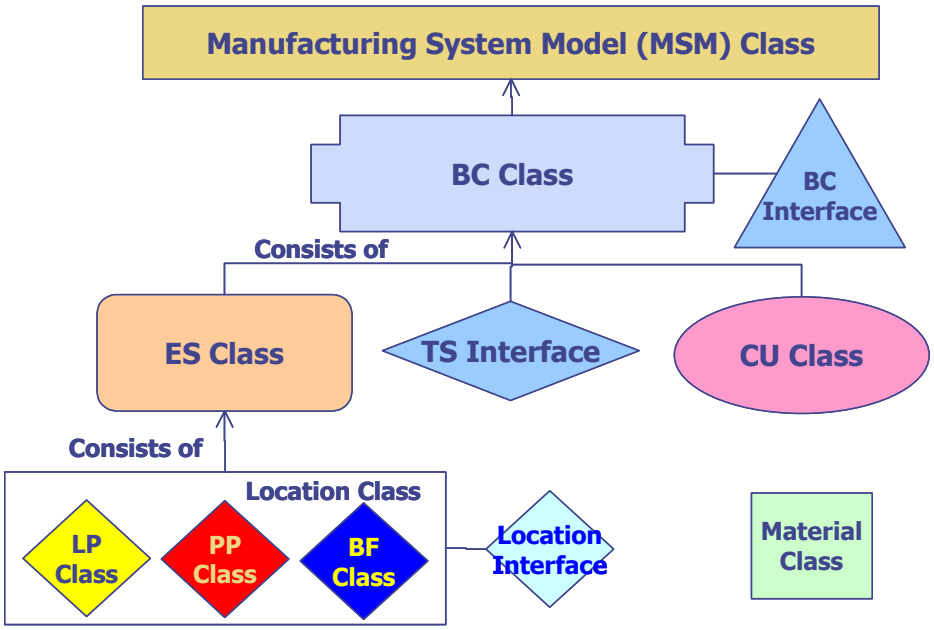


Figure 6.5 Abstract Class Template Model for RMM

One characteristic feature of the template model is that class templates have hierarchical relations with one another. For example, while the instances of Location Type Classes (LP Type Class, PP Type Class, BF Type Class) can be declared as components in ES Type class, the instance of ES Type Class cannot be declared as a component in any Location Type Classes. In other words, MSM Type Class consists of instances of BC Type Class, and BC Type Class also consists of its lower level instances of ES Type Class, TS Interface, and CU Class. Likewise, ES Type Class consists of instances of various Location Type Classes, and finally, the instances of Material Class are used in Location Type Class.

Here, we explain the features of these abstract class templates, and after defining the communication structure among classes, details of each class templates will be continued

in the following sections. For the convenience of explanation, we start from the lowest levels of abstract class templates.

Location Class (LP Class, PP Class, and BF Class) Template

The Location Class (LP Class, PP Class and BF Class) Template is a template to generate all classes for locations (load ports, process ports, and buffers) in the target system. As we explained in Table 5.1 in Chapter 5, in order to simulate location behaviors, the following attributes and behaviors should be defined.

Attributes:

LP Class

`Material _material;` //Status information of the material on load port

PP Class

`Material _material; State _state;` //Status information of the material and states of process port

BF Class

`Material[] _materials; Integer _capacity;` //Status information of the materials on the buffer, and size of the buffer

Behaviors:

Common Behaviors for LP, PP, and BF Class

```
Void LoadMaterial(Material mt); Material UnloadMaterial();
Boolean IsBlocked(); Material GetMaterial();
```

PP Class

```
Void StartProcess(State st); Void FinishProcess();
```

```
State GetState();
```

BF Class

```
Integer GetNumberOfLoadedMaterials();
Material GetMaterial(int index); Integer GetCapacity();
```

Each of these behaviors can be classified as one of three types; i) updating status information, ii) reporting status changes to control logic units (CUs), and iii) scheduling future events triggered by the behavior, as we explained in the simulation execution mechanism.

For updating status information, each of these behaviors has distinctive procedures. As an example, for LP Type Class, the fundamental procedures to update status information are defined as follows:

- `Void LoadMaterial(Material mt);`

  Assign loaded material `mt` to `_material`.

  ```
  { _material = mt; }
  ```

- `Material UnloadMaterial();`

  Return `_material` and make `_material` nullified.

  ```
  {Marterial tempMT = _material; _material = null; return tempMT;}
  ```

- `Boolean IsBlocked();`

  Check if there is any available space on the _material.

  ```
  { If _material != null {return true;} else { return false;} }
  ```

- `Material GetMaterial();`

Return the material information on _material.

```
{ return _material; }
```

Although updating status information procedures are common to all locations, other procedures such as reporting the status change and scheduling the future events differ according to the locations described in the RMM model.

Reporting the status change and scheduling the future events have the following formats:

- Format for Reporting Status Change

```
[CU name].UpdateInfo("[Loc name]","[Message]",_material.toString());
```

- Format for Schedule Future Event

```
Event ev = new Event([BC name], [EH name], [parameter]);
[SimObject].ScheduleEvent(ev, [SimObject].TNOW.getTime());
```

The contents between [and] are retrieved from the RMM model and filled in differently depending on location entities. For example, in the RMM model, if `cu1.DCI` = {<`lp1.LoadMaterial`, "*message description*">}, i.e. control logic unit, `cu1`, needs to get status update information from `lp1` for `LoadMaterial` behavior with "message description", then the following code should be inserted in `lp1.LoadMaterial()`.

```
Cu1.UpdateInfo(lp1, "message description", _material.toString());
```

Hence, to fill in the contents for Location Class, we need the following descriptions from RMM model.

- Names of locations in `LP`, `PP`, and `BF` of `ESM` model

- Descriptions in `CUM.DCI`
- Descriptions in `BCM.TE`

Location Interface

The Location Interface defines a set of common behaviors that should be implemented in every Location Type Class. This is very useful to handle various Location Type Classes in a higher-level class, because the instance of the interface can access any location class that implements Location Interface. In the following program,

```
{     LocationInterface _locInterface;
      LP1class lp1; PP2class pp2; BFclass bf;
      ….
      _locInterface = lp1; _locInterface.LoadMaterial(mt);
      _locInterface = pp2; _locInterface.GetMaterial();
      ….
},
```

because `_locInterface` is declared as a `LocationInterface`, it can assign any classes implemented the behaviors defined in Location Interface. Hence, in this sample program, because `lp1` and `pp2` are instances of Location Type Class that implements the behaviors of Location Interface, `_locInterface` can link with both `lp1` and `pp2`, and it can call behaviors implemented in `lp1` and `pp2` without any restrictions.

The behaviors of Location Interface are

- `Void LoadMaterial(Material mt);`
- `Material UnloadMaterial();`
- `Boolean IsBlocked();`

- `Material GetMaterial();`

As we mentioned, these behaviors should be implemented in all Location Type Classes (LP Type Class, PP Type Class, BF Type Class).

ES Class Template

The Equipment Structure Class (ES Class) Template is a template to generate the classes of equipment used in a target system. In RMM, equipment is defined by a set of locations and transport systems. We define the detailed specifications of the transport system in the equipment as a TS Interface separately. Therefore, ES Type Class deals purely with Location Type Classes. Since all locations in equipment are grouped in an ES Type class, this abstract class makes it convenient to handle the locations of equipment.

In RMM, all locations should be included in at least one of ES Type Class in accordance with the physical equipment, and some of the locations are included in more than one ES Type Class, which are called *shared locations*. Some `ESM` models in RMM, which consist of shared locations, are called *virtual equipment* that use the same formalism (`ESM`) for representing the physical systems but they model the sub-systems in the factory, such as work cells consisting of multiple physical equipment.

TS Interface

The Transport System Interface (TS Interface) defines a set of common behaviors that should be implemented in every Transport Controller Object. In RMM, Transport Systems exist as an external object (or Class) that implement behaviors defined in the TS Interface. The behaviors of the TS Interface are,

- `Void MoveOrder(Material mt, LocationInterface source, LocationInterface destination);`
- `Void LoadMaterial(Material mt, LocationInterface source);`
- `Material UnloadMaterial(LocationInterface destination);`

CU Class Template

The Control Logic Unit Class (CU Class) Template is a template to generate all classes of control logic units (CUs) designed in a target system. The main role of a CU is to represent decision-making behaviors that occur during manufacturing execution.

Since CU Type Class is related to the behaviors of Logical Modeling Entity explained in the previous section, its behaviors can be classified as one of three types: i) executing its decision algorithm, ii) updating internal data, when it receives a status update message from the physical modeling entities, and iii) scheduling future events that are driven by the execution of logical behaviors. Hence, the CU Class Template has the following structures for the above behaviors.

- Format for Declaring Internal Data Structure

```
[Class Name of CU.IDS] [Instance Name of CU.IDS] =
                                    new [Class Name of CU.IDS]();
```

- Format for Decision Making Procedure

```
[CU.DI.CUType] [CU.DI.CUInterface]([CU.DI.Parameters])
{
 [CU.DA, Program codes for decision algorithm]
};
```

- Format for Updating Internal Data

```
Void UpdateInfo(Message msg)
{
   if (msg.type = [CU.DCI:dci1.MEName, types of data update])
   {//message handling for dci1
   [CU.DCI:dci1.MPName, Program codes for internal data update];
   }
   else if ((msg.type = [CU.DCI:dci2.MEName, types of data update])
   {//message handling for dci1
   [CU.DCI:dci2.MPName, Program codes for internal data update];
   }
   ….
   Else{…};
}
```

- Format for Schedule Future Event

```
Event ev = new Event([BC name], [EH name], [parameter]);
[SimObject].ScheduleEvent(ev, [SimObject].TNOW.getTime());
```

The contents between [and] are retrieved from the RMM model and filled in differently according to the specific CUs. In order to fill in the contents for CU Type Class, we need the following descriptions from the RMM model.

- Names of CUs in CUM model
- Descriptions in CUM = <DI, IDS, DCI, DA>
- External classes or objects for CU.IDS, CU.DCI.MPName, and CU.DA.DAName

BC Class Template

The Behavior Coordinator Class (BC Class) Template is a template to generate the classes of behavior coordinators defined in an RMM model by representing behaviors of event handlers in the simulation execution system. Since the role of BC Type Class is the coordination of the components, which are represented as physical or logical modeling entities in the simulation execution mechanism, BC Class Template is configured with two parts: i) declarations of the instances of classes for physical or logical modeling entities, and ii) event handling behaviors for triggering events defined in the RMM model. Hence, the following shows the structure of BC Class Template reflecting the above explanations.

- Formats for Declaring Physical or Logical Modeling Classes

```
[Class Name of ES Class] [Instance Name of ES Class] =
                                    new [Class Name of ES Class]();
TSInterface [Instance Name of TS 1] = new [TSM:TS 1.TC]();
…
TSInterface [Instance Name n of TS n] = new [TSM:TS n.TC]();
[Class Name of CU Class 1] [Instance Name of CU Class 1] =
                                    new [Class Name of CU Class 1]();
….
[Class Name of CU Class m] [Instance Name of CU Class m] =
                                    new [Class Name of CU Class m]();
```

- Format for Trigger Event Handlers

```
Public void [BC.TE.te 1.EN]()
```

```
{
   [BC.TE.te 1.EH]
}
….
Public void [BC.TE.te k.EN]()
{
   [BC.TE.te k.EH]
}
```

The contents between [and] are retrieved from the RMM model and filled in differently according to the specific BCs. In order to fill in the contents for BC Type Class, we need the following descriptions in the RMM model.

- Name of BCs in BCM model
- Descriptions in BCM = <ESM, TSM, CUM, TE>
- Detail descriptions of BCM.TE = <EN, EH>
- External Classes for TSM.TC

BC Interface

The Behavior Coordinator Interface (BC Interface) defines a set of standard behaviors for interacting with the external transport controllers. The behaviors defined in BC Interface should be implemented in all BC Type Classes, and the behaviors of BC Interface are as follows.

- `Public void ReadyToLoad`
  `(TSInterface ts, LocationInterface destinationLoc);`
- `Public void ReadyToUnload`

```
(TSInterface ts, LocationInterface sourceLoc);
```

MSM Class Template

The Manufacturing System Model (MSM Class) Template is a template to generate the class of a target manufacturing system. Since we deal with a single manufacturing system in an RMM model, there is only a single corresponding MSM model in the simulation execution mechanism, and MSM Class represents the highest level of class including all physical and logical modeling entities. In RMM, a manufacturing system is configured with several BCM models. Hence, MSM Class also consists of several instances of BC Type Classes. The formats of MSM Class Template are as follows, and in order to fill the contents of the MSM Class Template, we need a list of behavior coordinators (BCM) defined in the RMM model.

- Formats for Declaring BC Type Classes

```
[Class Name of BC Class 1] [Instance Name of BC Class 1] =
                                    new [Class Name of BC Class 1]();
[Class Name of BC Class 2] [Instance Name of BC Class 2] =
                                    new [Class Name of BC Class 2]();
   ….
[Class Name of BC Class n] [Instance Name of BC Class n] =
                                    new [Class Name of BC Class n]();
```

Material Class

Materials are passive objects in RMM, which are moved or controlled by behavior coordinators (BCs). Hence, whatever the material is, it can be regarded just as an entity

providing status information such as `Type`, `Status`, and `ID`. Material Class is used to generate the instances of materials, which convey the information related to materials. As we defined it in Chapter 5, Material Class has attributes and behaviors as follows:

Attributes:

```
String _type, _status, _id;
```

Behaviors:

```
String GetType() { return _type };
String GetStatus() { return status };
String GetID() { return id };
Void SetType (String type) { type = type };
Void SetStatus(String status) { _status = status };
Void SetID(String id) { _id = id };
```

6.2.2 Communication of Classes

Executable simulation models in RMM consist of objects (instances of Class) driven by the Abstract Class Template illustrated in Figure 6.5. In order to execute a simulation model, each object should have communication links with other objects. In RMM, the communication links are classified into two types.

Direct Invoking

In OOP, in order for an object to call a behavior of another object, the calling object should have a reference pointer for the called object. As an illustration, in Figure 6.6, there are two objects: an instance (`a`) of Class A and an instance (`b`) of Class B. In order for `a` to call `b1`() behavior defined in object `b`, the reference pointer(`bR`) for object `b`

should be defined. After initialization to assign `b` to `bR` in object `a`, object `a` can communicate with object `b` through reference pointer `bR`.

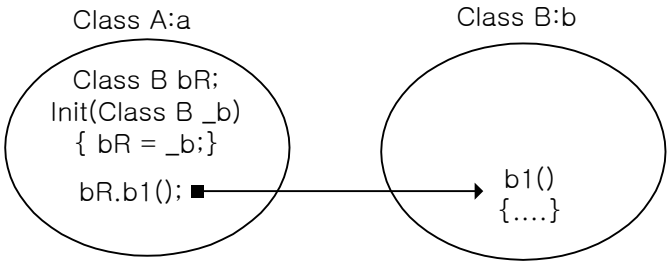


Figure 6.6 Illustration for Direct Invoking Communication Link

One of the characteristics in direct invoking communication link is that the simulation time is not changed during communication. Hence, the event handler uses direct invoking for communicating with physical and logical objects without advancing simulation time. However, to use direct invoking as a communication link when we design a class, we should know in advance what type of classes would be communicated with the class.

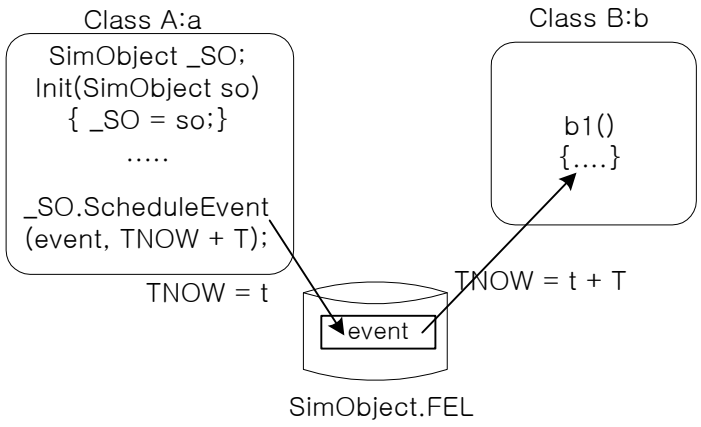


Figure 6.7 Illustration for Indirect Invoking Communication Link using FEL

Indirect Invoking

Another type of communication link is indirect invoking by using the future event list (`FEL`). As illustrated in Figure 6.7, when object `a` tries to use `b1()` behavior of object `b`, we can use the `FEL` defined in `SimObject` as a medium for communication. One of the benefits of this communication link is that we can make simulation time differences between the time a behavior is scheduled for and the time that it is actually executed. In other words, object `a` can schedule `b1()` behavior at simulation time (`TNOW`), `t`, but the actual execution of behavior can be performed at `t + T` as it is assigned in `ScheduleEvent()`. Although there is no direct link between calling and called objects, there should be direct invoking between the calling object and the `SimObject`.

Communication Requirements in SEM for RMM

Since an executable simulation model consists of multiple objects according to the Abstract Class Templates, we need to clarify the communication requirements based on the results of analysis of the simulation execution mechanism in section 6.1.

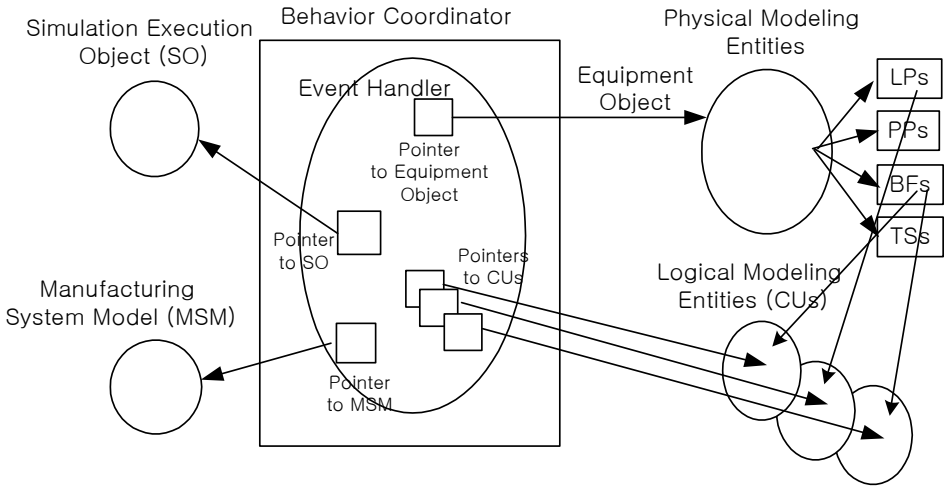


Figure 6.8 Communication Requirements in SEM for RMM

Figure 6.8 shows the required types of object for communications. A behavior coordinator (`BC Type Class`) needs direct invoking communication links with equipment object (`ES Type Class`), control logic units (`CU Type Class`), and the manufacturing system model (`MSM Class`) for linking with other behavior coordinators, and it also needs the indirect invoking communication link through the simulation execution object (`SO Class`). For updating information, `Location Type Class` requires direct invoking communication links with corresponding `CU Type Class`, and for scheduling future events, both `Location Type Class` and `CU Type Class` should have indirect invoking communication links. Table 6.1 shows the summary of communication requirements.

Table 6.1 Summary of Communication Requirements

	SO Class	MSM Class	BC Class	ES Class	Loc. Class	CU Class
SO Class		D				
MSM Class	D		D			
BC Class	ID	D		D		D
ES Class					D	
Loc. Class	ID					D
CU Class	ID					

D: Direct Invoking Com. Link, **I**: Indirect Invoking Com. Link

In order to reflect the communication requirements, we need to add program codes to link reference pointers for communication and to initialize communicating objects in designing the Abstract Class Templates. Since BC Class is the basic unit and MSM Class includes all instances of BC Type Class in RMM modeling, all physical or logical objects can communicate with one another by linking with instances of MSM Class and BC Type Classes.

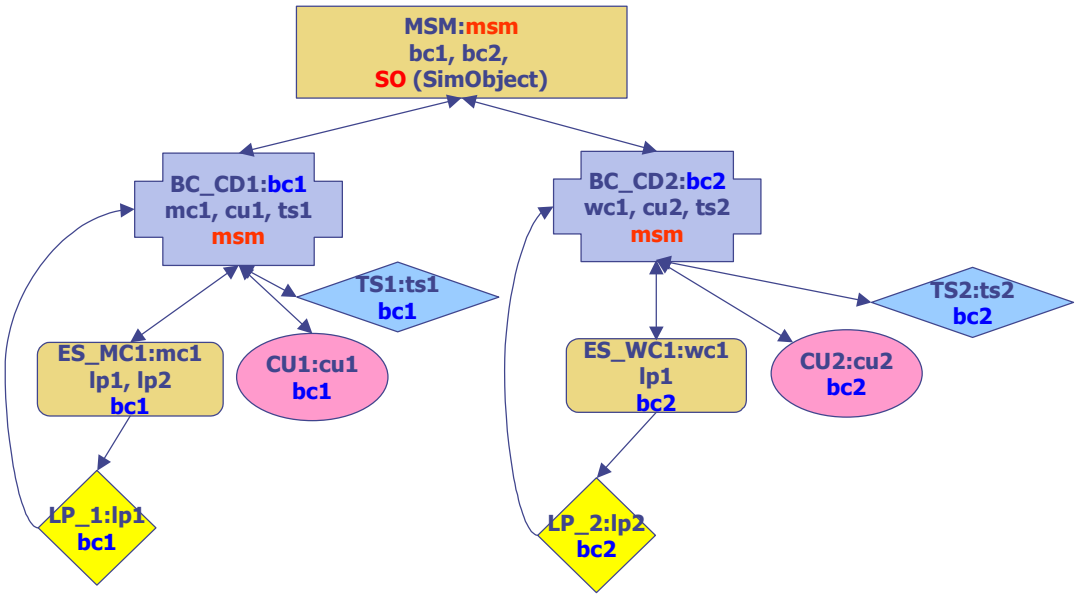


Figure 6.9 Illustration of Communication Links for RMM Model

Figure 6.9 illustrates the communication links among the instances in the RMM model. The MSM object (`msm`) has two BC objects (`bc1` and `bc2`), and has handshaking reference pointers for each other. In addition, BCs and other objects also have handshaking reference pointers. Hence, in this structure, all objects can communicate each other. For example, when `lp1` calls a behavior `x()` of `cu2` by the direct communication linking, it can be called by `lp1.bc1.msn.bc2.cu2.x()`, and when `lp1` schedules a trigger event(`te1`) of `bc2` by the indirect communication link, it can be called by `lp1.bc1.msm.SO.ScheduleNext()` with Event `ev = new Event (lp1.bc1.msn.bc2.te1, TNOW + t)`.

6.2.3 An Example of Abstract Class Template

As shown in Figure 6.9, the Abstract Class Template Model consists of eight Class Templates and three Interface Templates. The complete templates are presented in Appendix in detail. To design a template, we used imbedded tags interpreted by the *template interpreter*[12]. In this section, we explain the tags used in template modeling and show the MSM class template as an example.

```
1:Structure of Abstract Class Template
2:
3:Public class ClassName
4:{
5:    Class1 _instance1 = new Class1();
6:    Class2 _instance2 = new Class2();
7:    ....
8:    //Static program codes
9:
10:   //Tag
11:   <&
12:    //Dynamic program codes
13:    RS = SQL.EXE(.....
14:    ....
15:    ....
16:    ....
17:
18:    Print(ClassNa.....
19:   &>
20:
21:    void method()
22:    {
23:   <&
24:      //Dynamic program codes
25:   &>
26:     }
27:
28:}
```

Figure 6.10 Structure of Abstract Class Template

[12] The role of Template Interpreter is to generate simulation model classes after compiling the programming scripts within <& and &> tags. We can use VBScript or JavaScript compiler (Easttom 2001) for this purpose.

Figure 6.10 shows the structure of an Abstract Class Template. The statements in the template can be divided into two types: one is for the statements defined within tags, <& and &>, and the other is for the statements defined outside of the tags. The statements outside of the tags are the static statements that are copied and printed in a generated class without any changes. The statements within tags are program codes that should be interpreted by the template interpreter to generate the dynamic statements. Within tags, we can define variables and relevant functions that are used for generating dynamic statements. Since program classes are generated from the descriptions stored in the RMM database, we can use SQL syntax (Kline 2000) to access the contents of database inside of tags. After executing the program codes, the result is printed out into the class file by `Print()` function.

```
1:MSM_TP.Class Template
2:
3://Input: MSM := name of Manufacturing System Model
4:
5:Public class MSM
6:{
7:    SimObject _so = new SimObject();
8:
9:    //Declaring BC Type Classes Defined in BCM Table
10:   <&
11:    RS = SQL.EXE("SELECT * FROM BCM WHERE "BCM.MSMNAME='MSM'");
12:
13:    While (RS.NEXT())
14:    {
15:       STRING BC_NAME = RS.BCM_NAME;
16:       STRING ClassName = "BC_"+BC_NAME;
17:       STRING InstanceName = "_"+BC_NAME;
18:
19:       Print(ClassName+" "+InstanceName+" new "+ClassName+"();+\n");
20:    }
21:   &>
22:
23:    void init()
24:    {
25:   //Initialization
26:   <&
27:    RS = SQL.EXE("SELECT * FROM BCM WHERE BCM.MSMNAME='MSM'");
28:
29:    While (RS.NEXT())
30:    {
31:       STRING BC_NAME = RS.BCM_NAME;
32:       STRING InstanceName = "_"+BC_NAME;
33:
34:       Print(InstanceName+"(this, _so);\n");
35:    }
36:   &>
37:    }
38:
39:   void main()
40:   {
41:         MSM msn = new MSM();
42:
43:         msn.init();
44:         SO.ScheduleEvent(new Event("InitEvent",0.0);
45:   }
46:
47:}
```

Figure 6.11 Abstract Class Template for MSM Class

As an example, the MSM Class template is presented in Figure 6.11. In line (5), the static name of class is defined, and in line (11), it shows an SQL statement for extracting BC instances that are associated with the given MSM name and they are stored in a record set (RS). Using `while` loop in lines (13 ~ 20), it prints BC class instances in the MSM file. In order to create the dynamic statements for `init()` function, lines (27 ~ 36) are coded.

All the classes and instances are following the naming conventions as follows;

```
Class Name := "Type of Class Template"+"_"+"RMM Model Name",
              Instance Name := "_"+"RMM Model Name"
```

For example, if there is a load port named as "In1" in the RMM model, its class name is `LP_In1`, and its instance name in program is `_In1`.

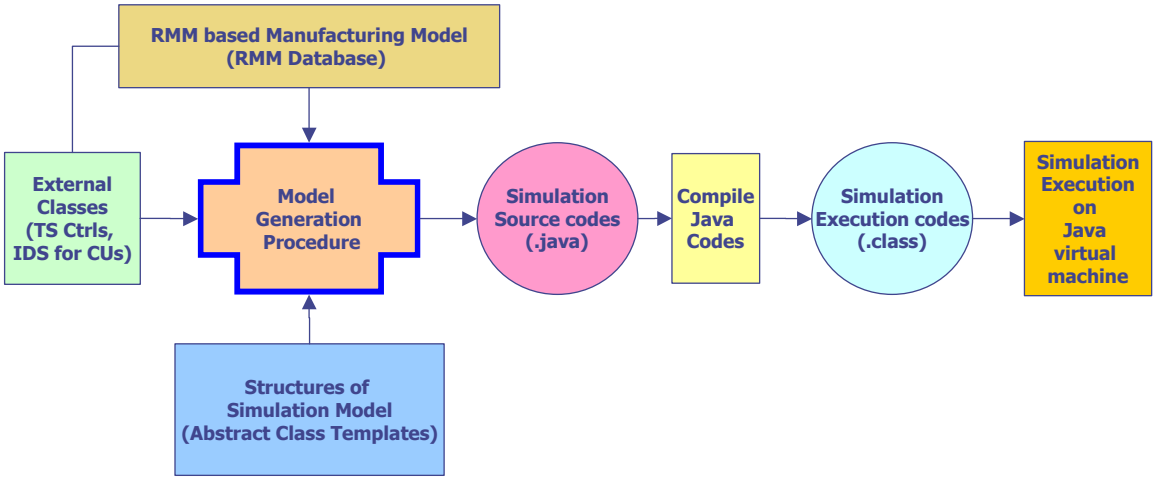


Figure 6.12 Flow of Automatic Generation of Simulation Models with Java

6.3 Automatic Model Generation Process

Since the abstract class templates are modeled with imbedded codes that can be interpreted by the *template interpreter*, we can generate executable simulation models by a single pass process. The detailed contents of the Model Generation Procedure are described in Appendix A.1.

Figure 6.12 shows the comprehensive flow of simulation in RMM. After a system is described with RMM formalism, an RMM model instance is stored in the RMM database with relevant external classes. Once a system is described and stored with RMM database structure, the Model Generation Procedure creates Java Class files that are used in the execution of simulation. After compiling Java Class files, they are converted to executable codes (.Class files), which can be executed on the Java Virtual Machine (JVM). Hence, the only manual process to generate executable simulation models is describing a target system in the form of RMM formalism.

If we develop a different template interpreter and Model Generation Procedure for different program languages, these can be plugged into this framework and generate simulation models with different program languages or simulation packages such as FORTRAN, C/C++, SIMAN, or GPSS.

6.4 Summary

In this chapter, we described the implementation framework in which an RMM model instance can be converted to an executable simulation model. As a basis execution

structure, we defined a simulation execution mechanism for RMM that matches the RMM structure. Based on the results of requirement analysis, we designed the Abstract Class Template model in which RMM model instances can be realized as executable program codes. Due to the formal structure of the Abstract Class Template, we were able to efficiently build a model generation procedure for generating simulation models automatically.

CHAPTER VII

CONCLUSIONS

A high fidelity simulation model is important not only for the accuracy in simulation results, but also for simulation modeling productivity. For a given target system, if we can build a high fidelity model (the so-called virtual factory) representing system behaviors in detail, various simulation models can be derived. These simulation models with various purposes can be derived from the high fidelity model through simple structural and behavioral abstractions, instead of building abstracted simulation models from scratch whenever new situations arise. Since system knowledge is imbedded in the high fidelity model, we can systematically reuse the results of system analysis as well as simulation program codes. This modeling methodology is called the Virtual Factory Approach. If we can build such a high fidelity model efficiently, the Virtual Factory Approach can be an alternative for improving simulation modeling productivity.

However, there are obstacles in deploying the Virtual Factory Approach. The first one is a theoretical question as to whether the simulation models derived from a high fidelity model are behaviorally equivalent to (or interchangeable with) simulation models built from scratch. Although the idea of equivalence in simulation models is conceptually simple, it is very difficult to compare without actually having to run both models under all possible experimental conditions and compare their output results. The second obstacle is the difficulty in developing a virtual factory with current simulation modeling methodologies. As the degree of fidelity is increased, modeling complexity also gets

increased. Current modeling tools do not provide a systematic method to cope with the complexity of building high fidelity simulation models.

7.1 Conclusions and Research Contributions

As Yücesan and Schruben (1992) mentioned, it is highly unlikely in practice to determine whether simulation models are behaviorally equivalent or not without experiments and output analysis. Hence, in this research, we opted to use the fidelity of models as a surrogate metric. Since fidelity is a metric for measuring the closeness to a real system, if models have same fidelity and it is measurable without actual experiments, we can regard them as interchangeable. Unfortunately, it is also very difficult to measure the fidelity of models in an absolute and quantitative way (Gross *et al*. 1999).

In this thesis, we developed the relative fidelity indicator for comparing models and developed a systematic way for determining the equivalence of simulation models. The relative fidelity is based on the derivability relations between experimental frames, which are defined as input and output variables used in simulation models. If an experimental frame for a model is derivable from that of another model, we can say that the model from which the experimental frame is derived has lower fidelity than other models. In addition, by using the relative fidelity indicator and simulation modeling framework, we identified a theoretical property–that the higher fidelity model has higher reusability in simulation modeling. We also classified various abstraction methods that enable modelers to derive various abstract simulation models from a virtual factory.

For simulation modeling, instead of focusing on a specific target system, we expanded our focus onto a domain of similar systems in this thesis. Through the analysis of the domain, we could find ways to describe any system in the domain and to structure domain knowledge in order to reuse it for modeling systems in the domain. We could automatically generate executable simulation models for any system in the domain from the formal system descriptions. This new simulation modeling methodology is called the Reference Model Methodology. We developed a general procedure for the Reference Model Methodology and identified its characteristics.

For a specific domain, two fundamental elements for successful deployment of the Reference Model Methodology are a *reference model* and a *model generation procedure*. The reference model is a formal structure for describing systems in a domain, and the model generation procedure is a program to generate executable simulation models automatically from formal descriptions by incorporating structuralized domain knowledge. The reference model and the model generation procedure can be developed through domain analysis. Domain analysis is a process to find common modeling entities and logical rules that characterize and control the behaviors of systems in the target domain.

In this thesis, we applied the Reference Model Methodology to a specific domain of discrete part manufacturing systems. From the perspective of material flow and control, we found a necessary and sufficient set of physical, logical and coordinating modeling entities through domain analysis, and formalized modeling entities as a reference model

in the forms of relational database schema. Based on this reference model for manufacturing (RMM), we developed a model generation procedure using a set of abstract class templates, which are skeleton programs incorporating common domain knowledge.

In this research, we contribute to the simulation modeling for manufacturing systems as follows

- Developing a formal simulation modeling framework for comparing the fidelity of simulation models, and deriving characteristic relationships between simulation models and model fidelity.
- Developing and formalizing the Reference Model Methodology, which is an efficient simulation modeling methodology for a specific domain, and deploying the methodology onto the discrete part manufacturing systems.

7.2 Future Research

The Reference Model Methodology can be applied to those domains of which domain knowledge and descriptions can be formalized. This methodology has been developed through the experiences in the research project HiFiVE (High Fidelity Virtual Environment for Manufacturing Systems) at the Keck Virtual Factory Lab in Georgia Tech (Kim *et al*. 2000, 2001, 2002). We can also consider various other domains such as warehousing systems, supply chain management systems, and transportation systems. In deploying the methodology on these domains, we need methodologies for domain analysis to create a reference model and a model generation procedure for the particular

domain. Because the methodologies for domain analysis are different depending on the domain characteristics and the results of domain analysis have big a impact on the quality of simulation modeling, we need more research for analyzing the domain correctly for the requirements and in evaluating the quality of the results of domain analysis.

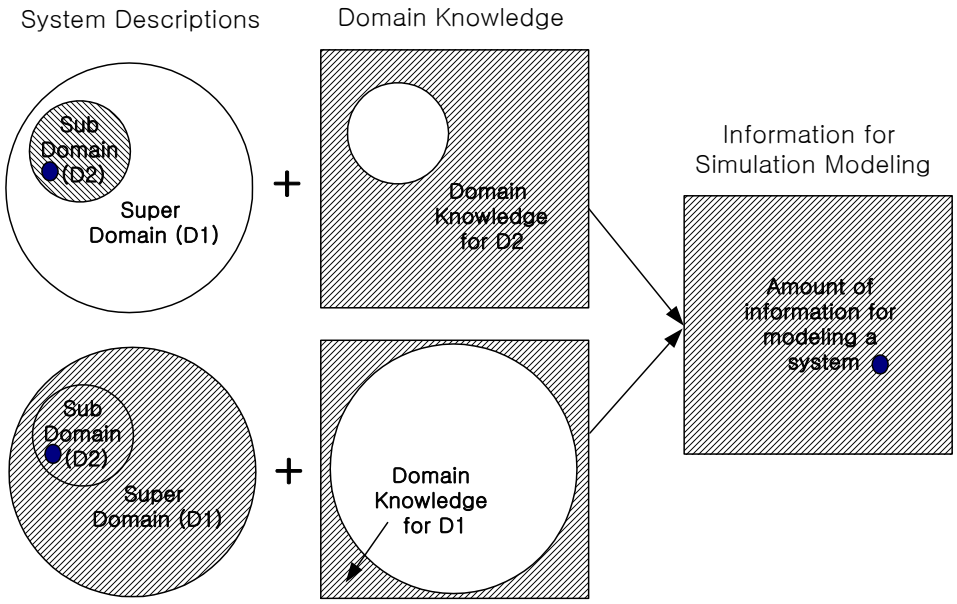


Figure 7.1 Relationship between Super Domain (D1) and Sub Domain (D2)

From a modeling theory point of view, it is also important to study the relationships between super and sub domains. While super domain relies more on the formal descriptions for generating simulation models, some of the domain knowledge represented as formal descriptions in the super domain are imbedded in the model generation procedure of sub domain as illustrated in Figure 7.1. In practice, if there is a system included in both super and sub domains, it is more efficient to use the sub domain's reference model and model generation procedure in modeling because it

requires less modeling descriptions than the super domain. Therefore, when there is a set, of a reference model and a model generation procedure for the super domain, it makes for interesting research to see how to reuse the existing results of the super domain for deriving the reference model and the model generation procedure of the sub domain efficiently and systematically, as in the Virtual Factory Approach to derive the abstracted simulation model from a high fidelity simulation model.

In this thesis, we focused only on the improvement of simulation modeling productivity. For simulating a high fidelity simulation model, it requires high computing environments. Hence, it is important to consider simulation execution efficiency as well. In order to increase the simulation computing power, there has been a lot of research on parallel and distributed simulation (PADS) executive systems. Therefore, it is also a challenging and interesting research topic to improve the Reference Model Methodology pursuing efficient modeling productivity as well as execution productivity by using a parallel and distributed simulation execution environment.

APPENDIX

PROGRAMS FOR MODEL GENERATION PROCEDURE AND ABSTRACT CLASS TEMPLATES FOR RMM

A.1 Model Generation Procedure

```
1   //Model Generation Procedure
2
3   Public Class MGP
4   {
5         void main()
6         {
7               String msmName = new String("/*put the name of MSM*/");
8
9                     LOC_Class_Generation();
10                    ES_Class_Generation();
11                    CU_Class_Generation();
12                    BC_Class_Generation();
13                    MSM_Class_Generation(msmName);
14                    InterfaceGeneration();
15                    Material_Class_Generation();
16        }
17
18        public void LOC_Class_Generation()
19        {
20           RS = SQL.EXE("SELECT * FROM ESM_LOCATION ORDERED BY
21  LOC_NAME");
22
23         while(RS.NEXT())
24         {
25                  if(RS.TYPE = "LP")
26                   TemplateInterpreter(LP_TP, RS.LOC_NAME);
27                  else if(RS.TYPE = "PP")
28                   TemplateInterpreter(PP_TP, RS.LOC_NAME);
29                  else if (RS.TYPE = "BF")
30                   TemplateInterpreter(BF_TP, RS.LOC_NAME);
31                  else
32                  {}
33              }
34        }
35
36        public void ES_Class_Generation()
37        {
38              RS = SQL.EXE("SELECT * FROM ESM ORDERED BY ESM_NAME");
39
40              while(RS.NEXT())
```

```
41                {
42                      TemplateInterpreter(ES_TP, RS.ESM_NAME);
43                }
44          }
45
46          public void CU_Class_Generation()
47          {
48                RS = SQL.EXE("SELECT * FROM CUM ORDERED BY CUM_NAME");
49
50                while(RS.NEXT())
51                {
52                      TemplateInterpreter(CU_TP, RS.CUM_NAME);
53                }
54          }
55
56        public void BC_Class_Generation()
57        {
58                RS =SQL.EXE("SELECT * FROM BCM ORDERED BY BCM_NAME");
59
60                while(RS.NEXT())
61                {
62                      TemplateInterpreter(BC_TP, RS.BCM_NAME);
63                }
64
65          }
66
67          public void MSM_Class_Generation(String msm_name)
68          {
69                TemplateInterpreter(MSM_TP, msm_name);
70          }
71
72          public void InterfaceGeneration()
73          {
74                BCInterface_Generation();
75                TSInterface_Generation();
76                LocationInterface_Generation();
77          }
78
79          public void Material_Class_Generation()
80          {
81                Material_Generation();
82          }
83
84    }
```

A.2 Class Templates

MSM_TP

```
1  MSM_TP.Class Template
2
3  //Input: MSM := name of Manufacturing System Model
4
5  Public class MSM
6  {
7        SimObject _so = new SimObject();
8
9        //Declaring BC Type Classes Defined in BCM Table
10       <&
11        RS = SQL.EXE("SELECT * FROM BCM WHERE "BCM.MSMNAME='MSM'");
12
13        While (RS.NEXT())
14        {
15           STRING BC_NAME = RS.BCM_NAME;
16           STRING ClassName = "BC_"+BC_NAME;
17           STRING InstanceName = "_"+BC_NAME;
18
19           Print(ClassName+" "+InstanceName+" new "+ClassName+"();+\n");
20        }
21       &>
22
23     void init()
24     {
25       //Initialization
26       <&
27        RS = SQL.EXE("SELECT * FROM BCM WHERE BCM.MSMNAME='MSM'");
28
29        While (RS.NEXT())
30        {
31           STRING BC_NAME = RS.BCM_NAME;
32           STRING InstanceName = "_"+BC_NAME;
33
34           Print(InstanceName+"(this, _so);\n");
35        }
36       &>
37     }
38
39       void main()
40       {
41             MSM msn = new MSM();
42
43             msn.init();
44             SO.ScheduleEvent(new Event("InitEvent",0.0);
45       }
46
```

BC_TP

```
1   BC_TP.Class Template
2
3   //Input: BC_Name := name of BC model
4
5   Public class BC_<& BC_Name &> implementation BCInterface
6   {
7         SimObject _so = new SimObject();
8         MSM _msm = new MSM();
9
10      //Declaring ES, CU, Type Classes and TS Interfaces
11      <&
12       RS = SQL.EXE("SELECT * FROM ESM WHERE ESM.BCM_NAME='BC_Name'");
13
14       //Declairing ES Class
15       While (RS.NEXT())
16       {
17               STRING ES_NAME = RS.ESM_NAME;
18               STRING ClassName = "ES_"+ES_NAME;
19               STRING InstanceName = "_"+ES_NAME;
20
21               Print(ClassName+" "+InstanceName+" new
22  "+ClassName+"();+\n");
23          }
24
25          //Declairing TS Interface
26           RS = SQL.EXE("SELECT * FROM TSM WHERE TSM.BCM_NAME='BC_Name'");
27
28           While (RS.NEXT())
29           {
30                STRING TS_NAME = RS.TC_NAME;
31                STRING ClassName = "TS_"+TS_NAME;
32                STRING InstanceName = "_"+TS_NAME;
33
34                Print(ClassName+" "+InstanceName+" new
35  "+ClassName+"();+\n");
36          }
37
38
39          //Declairing CU Class
40           RS = SQL.EXE("SELECT * FROM CUM WHERE CUM.BCM_NAME='BC_Name'");
41
42           While (RS.NEXT())
43           {
44                STRING CU_NAME = RS.CUM_NAME;
45                STRING ClassName = "CU_"+CU_NAME;
46                STRING InstanceName = "_"+CU_NAME;
47
48                Print(ClassName+" "+InstanceName+" new
49  "+ClassName+"();+\n");
50          }
51      &>
52
53      void init(MSM msm, SimObject so)
54      {
```

```
55              _msm = msm;
56              _so = so;
57
58        //Initialization
59      <&
60       RS = SQL.EXE("SELECT * FROM ESM WHERE ESM.BCM_NAME='BC_Name'");
61
62       //Initialization for ES Class
63       While (RS.NEXT())
64       {
65               STRING InstanceName = "_"+RS.ESM_NAME;
66               Print(InstanceName+".init(this);");
67         }
68
69         //Initialization for TS Interfaces
70          RS = SQL.EXE("SELECT * FROM TSM WHERE TSM.BCM_NAME='BC_Name'");
71
72          While (RS.NEXT())
73          {
74               STRING InstanceName = "_"+RS.TC_NAME;
75               Print(InstanceName+".init("+BC_Name+");");
76          }
77
78         //Initialization for CU Interfaces
79          RS = SQL.EXE("SELECT * FROM CUM WHERE CUM.BCM_NAME='BC_Name'");
80
81          While (RS.NEXT())
82          {
83               STRING InstanceName = "_"+RS.CUM_NAME;
84               Print(InstanceName+".init(this);");
85          }
86       &>
87
88      }
89
90        //Trigger Event Handlers
91
92      <&
93          RS = SQL.EXE("SELECT * FROM BCM_TE WHERE
94  BCM_TE.BCM_NAME='BC_Name'");
95
96          While (RS.NEXT())
97          {
98               STRING EventName = RS.EN_NAME;
99               STRING EventHandler = File(RS.EH_NAME).getText();
100              Print("Public void "+EventName+
101                      "{"+ EventHandler +"} \n");
102         }
103      &>
104
105 }
106
```

ES_TP

```
1   ES_TP.Class Template
2
3   //Input: ES_Name := name of ES model
4
5   Public class ES_<& ES_Name &>
6   {
7
8       //Declairing linked BC Class
9       <&
10       RS = SQL.EXE("SELECT * FROM ESM WHERE ESM.ESM_NAME='ES_Name'");
11
12       RS.NEXT();
13          STRING BC_NAME = RS.BCM_NAME;
14          STRING ClassName = "BC_"+BC_NAME;
15          STRING InstanceName = "_"+BC_NAME;
16       Print(ClassName+" "+InstanceName+" new "+ClassName+"();+\n");
17      &>
18
19         //Declairing Location Class
20      <&
21          RS = SQL.EXE("SELECT * FROM ESM_LOCATION WHERE
22              ESM_LOCATION.ESM_NAME='ES_Name'");
23
24          While (RS.NEXT())
25          {
26               STRING LOC_NAME = RS.LOC_NAME;
27               STRING ClassName;
28
29               if (RS.TYPE = "LP")
30               {
31                     ClassName = "LP_"+ LOC_NAME;
32               }
33               else if (RS.TYPE = "PP")
34               {
35                     ClassName = "PP_"+ LOC_NAME;
36               }
37               else if (RS.TYPE = "BF")
38               {
39                     ClassName = "BF_"+ LOC_NAME;
40               }
41               else
42               {}
43
44               STRING InstanceName = "_"+LOC_NAME;
45               Print(ClassName+" "+InstanceName+" new
46  "+ClassName+"();+\n");
47         }
48        &>
49
50      void init(<& Print("BC_"+BC_NAME+" "+BC_NAME) &>)
51        {
52          <& Print ("_"+BC_NAME = BC_NAME) &>;
53
54        //Initialization of Locations
```

```
55        <&
56            RS = SQL.EXE("SELECT * FROM ESM_LOCATION WHERE
57                ESM_LOCATION.ESM_NAME='ES_Name' ");
58
59            While (RS.NEXT())
60            {
61                 STRING LOC_NAME = RS.LOC_NAME;
62                 STRING InstanceName = "_"+LOC_NAME;
63
64                 Print(InstanceName+".init(_"+BC_NAME+");");
65            }
66          &>
67          }
68    }
```

CU_TP

```
1   CU_TP.Class Template
2
3   //Input: CU_Name := name of CU model
4
5   Public class CU_<& CU_Name &>
6   {
7
8       //Declairing linked BC Class and Declairing Internal Data Strucutre
9       <&
10       RS = SQL.EXE("SELECT * FROM CUM WHERE CUM.CUM_NAME='CU_Name'");
11
12       RS.NEXT();
13          STRING BC_NAME = RS.BCM_NAME;
14          STRING ClassName = "BC_"+BC_NAME;
15          STRING InstanceName = "_"+BC_NAME;
16       Print(ClassName+" "+InstanceName+" new "+ClassName+"();+\n");
17
18          STRING ClassName = "CU_"+RS.IDS_NAME;
19          STRING InstanceName = "_"+RS.IDS_NAME;
20       Print(ClassName+" "+InstanceName+" new "+ClassName+"();+\n");
21        &>
22
23
24      void init(<& Print("BC_"+BC_NAME+" "+BC_NAME) &>)
25        {
26          <& Print ("_"+BC_NAME = BC_NAME) &>;
27        }
28
29
30        //Definitions of Decision Interfaces
31        <&
32         RS = SQL.EXE("SELECT * FROM CUM_DI WHERE
33  CUM_DI.CUM_NAME='CU_Name'");
34
35         While (RS.NEXT())
36         {
37              STRING DI_NAME = RS.DI_NAME;
38              STRING DI_TYPE = RS.TYPE;
39              STRING PARAMET = RS.Parameter;
40
41              Print("Public "+DI_TYPE+" "+DI_NAME+"("+PARAMET+")");
42
43              RS2 = SQL.EXE("SELECT * FROM CUM WHERE CUM.CUM_NAME
44  ='CU_Name'");
45              RS2.NEXT();
46              STRING DecisionAlgorithm = File(RS.DA_NAME).getText();
47             Print( "{"+ DecisionAlgorithm +"} \n");
48        }
49
50        <&
51         Print("Public void UpdateInfo(Message msg)");
52         Print("{\n");
53
```

```
54                RS = SQL.EXE("SELECT * FROM CUM_DCI WHERE
55   CUM_DCI.CUM_NAME='CU_Name'");
56                RS.NEXT();
57
58                Print("if(msg.type) ==" + "'RS.EVENTNAME'");
59                Print("{"+RS.HANDLER+"}");
60
61                While (RS.NEXT())
62                {
63                      Print("else if(msg.type) ==" + "'RS.EVENTNAME'");
64                      Print("{"+RS.HANDLER+"}");
65                }
66
67                Print("else");
68                Print("{ }");
69
70           Print("}\n");
71       &>
72   }
```

LP_TP

```
LP_TP.Class Template

//Input: LP_Name := name of LP model

Public class LP_<& LP_Name &> implement LocationInterface
{
    //Declairing linked BC Class
    <&
     RS = SQL.EXE("SELECT * FROM ESM_LOCATION WHERE
ESM.LOC_NAME='LP_Name'");
     RS.NEXT();

     STRING ES_Name = RS.ESM_NAME;

     RS2 = SQL.EXE("SELECT * FROM ESM WHERE ESM.ESM_NAME='ES_Name'");

       STRING BC_NAME = RS2.BCM_NAME;
       STRING ClassName = "BC_"+BC_NAME;
       STRING BCInstanceName = "_"+BC_NAME;
     Print(ClassName+" "+BCInstanceName+" new "+ClassName+"();+\n");
    &>

    Material _material;

    void init(<& Print("BC_"+BC_NAME+" "+BC_NAME) &>)
      {
        <& Print ("_"+BC_NAME = BC_NAME + ";") &>;
      }

    Public void LoadMaterial(Material mt)
    {
            _material = mt;

        <&
         RS2 = SQL.EXE("SELECT * FROM CUM_DCI");

         While(RS2.NEXT())
         {
             if(RS2.EVENTNAME == "LP_"+LP_NAME+".LoadMaterial")
             {
               Print(BCInstanceName+".CU_"+RS2.CUM_NAME+
                 ".UpdateInfo("+RS2.MESSAGE+");")
             }
         }

        RS3 = SQL.EXE("SELECT * FROM BCM_TE");

        While(RS3.NEXT())
        {
              if (RS3.EN_NAME == "_"+LP_NAME+".ENTER")
              {
                  Print("Event ev = new Event("+RS3.EN_NAME+");");
```

```
53
54          Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
55      ime());");
56                      }
57
58            }
59          }
60
61
62          Public Material UnLoadMaterial()
63          {
64                  Material mt = _material;
65
66           <&
67             RS4 = SQL.EXE("SELECT * FROM CUM_DCI");
68
69             While(RS4.NEXT())
70             {
71                 if(RS4.EVENTNAME == "LP_"+LP_NAME+".UnLoadMaterial")
72                 {
73                   Print(BCInstanceName+".CU_"+RS4.CUM_NAME+
74                     ".UpdateInfo("+RS4.MESSAGE+");")
75                 }
76             }
77
78            RS5 = SQL.EXE("SELECT * FROM BCM_TE");
79
80            While(RS5.NEXT())
81            {
82                  if (RS5.EN_NAME == "_"+LP_NAME+".LEAVE")
83                  {
84                      Print("Event ev = new Event("+RS5.EN_NAME+");");
85
86          Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
87      ime());");
88                      }
89
90            }
91                 Print("return mt;\n");
92           &>
93          }
94
95          Public boolean IsBlocked()
96          {
97                  if (_material != null)
98                  { return true; }
99                  else
100                 { return false;}
101
102         }
103
104         Public Material GetMaterial()
105         {
106               return _material;
107         }
108
109     }
```

PP_TP

```
1   PP_TP.Class Template
2
3   //Input: PP_Name := name of PP model
4
5   Public class PP_<& PP_Name &> implement LocationInterface
6   {
7       //Declairing linked BC Class
8       <&
9        RS = SQL.EXE("SELECT * FROM ESM_LOCATION WHERE
10  ESM.LOC_NAME='PP_Name'");
11       RS.NEXT();
12
13       STRING ES_Name = RS.ESM_NAME;
14
15       RS2 = SQL.EXE("SELECT * FROM ESM WHERE ESM.ESM_NAME='ES_Name'");
16
17         STRING BC_NAME = RS2.BCM_NAME;
18         STRING ClassName = "BC_"+BC_NAME;
19         STRING BCInstanceName = "_"+BC_NAME;
20       Print(ClassName+" "+BCInstanceName+" new "+ClassName+"();+\n");
21      &>
22
23      Material _material;
24      Status   _status;
25
26      void init(<& Print("BC_"+BC_NAME+" "+BC_NAME) &>)
27        {
28          <& Print ("_"+BC_NAME = BC_NAME + ";") &>;
29        }
30
31      Public void LoadMaterial(Material mt)
32      {
33              _material = mt;
34
35          <&
36           RS2 = SQL.EXE("SELECT * FROM CUM_DCI");
37
38           While(RS2.NEXT())
39           {
40               if(RS2.EVENTNAME == "PP_"+PP_NAME+".LoadMaterial")
41               {
42                 Print(BCInstanceName+".CU_"+RS2.CUM_NAME+
43                   ".UpdateInfo("+RS2.MESSAGE+");")
44               }
45           }
46
47          RS3 = SQL.EXE("SELECT * FROM BCM_TE");
48
49          While(RS3.NEXT())
50          {
51                if (RS3.EN_NAME == "_"+PP_NAME+".ENTER")
52                {
53                    Print("Event ev = new Event("+RS3.EN_NAME+");");
```

```
54
55          Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
56      ime());");
57                   }
58
59            }
60          }
61
62
63          Public Material UnLoadMaterial()
64          {
65                  Material mt = _material;
66
67           <&
68             RS4 = SQL.EXE("SELECT * FROM CUM_DCI");
69
70             While(RS4.NEXT())
71             {
72                 if(RS4.EVENTNAME == "PP_"+PP_NAME+".UnLoadMaterial")
73                 {
74                   Print(BCInstanceName+".CU_"+RS4.CUM_NAME+
75                     ".UpdateInfo("+RS4.MESSAGE+");")
76                 }
77             }
78
79            RS5 = SQL.EXE("SELECT * FROM BCM_TE");
80
81            While(RS5.NEXT())
82            {
83                  if (RS5.EN_NAME == "_"+PP_NAME+".LEAVE")
84                  {
85                      Print("Event ev = new Event("+RS5.EN_NAME+");");
86
87          Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
88      ime());");
89                   }
90
91            }
92              Print("_status = null;");
93                Print("return mt;\n");
94           &>
95          }
96
97          Public void StartProcess(State st):
98          {
99           <&
100           RS6 = SQL.EXE("SELECT * FROM CUM_DCI");
101
102           While(RS6.NEXT())
103           {
104               if(RS5.EVENTNAME == "PP_"+PP_NAME+".StartProcess")
105               {
106                 Print(BCInstanceName+".CU_"+RS6.CUM_NAME+
107                   ".UpdateInfo("+RS6.MESSAGE+");")
108               }
109           }
110
```

```
111           RS7 = SQL.EXE("SELECT * FROM BCM_TE");
112
113           While(RS7.NEXT())
114           {
115                 if (RS7.EN_NAME == "_"+PP_NAME+".Start")
116                 {
117                     Print("Event ev = new Event("+RS7.EN_NAME+");");
118
119         Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
120   ime());");
121                 }
122
123           }
124             Print("_status ="+st+";\n");
125          &>
126
127         }
128
129
130         Public void FinishProcess(State st):
131         {
132          <&
133            RS8 = SQL.EXE("SELECT * FROM CUM_DCI");
134
135            While(RS8.NEXT())
136            {
137                if(RS8.EVENTNAME == "PP_"+PP_NAME+".FinishProcess")
138                {
139                  Print(BCInstanceName+".CU_"+RS8.CUM_NAME+
140                    ".UpdateInfo("+RS8.MESSAGE+");")
141                }
142            }
143
144           RS9 = SQL.EXE("SELECT * FROM BCM_TE");
145
146           While(RS9.NEXT())
147           {
148                 if (RS9.EN_NAME == "_"+PP_NAME+".Finish")
149                 {
150                     Print("Event ev = new Event("+RS9.EN_NAME+");");
151
152         Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
153   ime());");
154                 }
155
156           }
157             Print("_status ="+st+";\n");
158          &>
159
160         }
161
162         Public boolean IsBlocked()
163         {
164                 if (_material != null)
165                 { return true; }
166                 else
167                 { return false;}
```

```
168
169         }
170
171         Public Material GetMaterial()
172         {
173               return _material;
174         }
175
176         Public Status GetStatus()
177         {
178               return _status;
179         }
180
181
182   }
```

BF_TP

```
1   BF_TP.Class Template
2
3   //Input: BF_Name := name of BF model
4
5   Public class BF_<& BF_Name &> implement LocationInterface
6   {
7       //Declairing linked BC Class
8       <&
9        RS = SQL.EXE("SELECT * FROM ESM_LOCATION WHERE
10  ESM.LOC_NAME='BF_Name'");
11       RS.NEXT();
12
13       STRING ES_Name = RS.ESM_NAME;
14       STRING OP_Policy = RS.OP_POLICY;
15
16       RS2 = SQL.EXE("SELECT * FROM ESM WHERE ESM.ESM_NAME='ES_Name'");
17
18         STRING BC_NAME = RS2.BCM_NAME;
19         STRING ClassName = "BC_"+BC_NAME;
20         STRING BCInstanceName = "_"+BC_NAME;
21       Print(ClassName+" "+BCInstanceName+" new "+ClassName+"();+\n");
22
23      &>
24
25      MaterialQueue _material;
26      Integer       _capacity;
27
28      <&
29        if (OP_Policy == "FIFO")
30        {
31                _matgerial = new MaterialQueue(_capacity, "FIFO");
32          }
33          else if (OP_Policy == "LIFO")
34          {
35                _matgerial = new MaterialQueue(_capacity, "LIFO");
36          }
37          else
38          {
39                _material = new MAterialQueue(_capacity, null);
40        }
41      &>
42
43      void init(<& Print("BC_"+BC_NAME+" "+BC_NAME) &>)
44        {
45          <& Print ("_"+BC_NAME = BC_NAME + ";") &>;
46        }
47
48      Public void LoadMaterial(Material mt)
49      {
50
51          _material.loadLast(mt);
52
53          <&
54           RS2 = SQL.EXE("SELECT * FROM CUM_DCI");
```

```
55
56           While(RS2.NEXT())
57           {
58               if(RS2.EVENTNAME == "BF__"+BF_NAME+".LoadMaterial")
59               {
60                 Print(BCInstanceName+".CU_"+RS2.CUM_NAME+
61                   ".UpdateInfo("+RS2.MESSAGE+");")
62               }
63           }
64
65          RS3 = SQL.EXE("SELECT * FROM BCM_TE");
66
67          While(RS3.NEXT())
68          {
69                if (RS3.EN_NAME == "_"+BF_NAME+".ENTER")
70                {
71                    Print("Event ev = new Event("+RS3.EN_NAME+");");
72
73        Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
74    ime());");
75                }
76
77          }
78        }
79
80
81      Public void LoadMaterial(Material mt, Integer idx)
82      {
83
84          _material.loadAt(idx,mt);
85
86          <&
87           RS2 = SQL.EXE("SELECT * FROM CUM_DCI");
88
89           While(RS2.NEXT())
90           {
91               if(RS2.EVENTNAME == "BF_"+BF_NAME+".LoadMaterial")
92               {
93                 Print(BCInstanceName+".CU_"+RS2.CUM_NAME+
94                   ".UpdateInfo("+RS2.MESSAGE+");")
95               }
96           }
97
98          RS3 = SQL.EXE("SELECT * FROM BCM_TE");
99
100         While(RS3.NEXT())
101         {
102               if (RS3.EN_NAME == "_"+BF_NAME+".ENTER")
103               {
104                   Print("Event ev = new Event("+RS3.EN_NAME+");");
105
106       Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
107   ime());");
108               }
109
110         }
111       }
```

```
112
113
114       Public Material UnLoadMaterial()
115       {
116               Material mt = _material.unloadFirst();
117
118          <&
119            RS4 = SQL.EXE("SELECT * FROM CUM_DCI");
120
121            While(RS4.NEXT())
122            {
123                if(RS4.EVENTNAME == "BF_"+BF_NAME+".UnLoadMaterial")
124                {
125                  Print(BCInstanceName+".CU_"+RS4.CUM_NAME+
126                    ".UpdateInfo("+RS4.MESSAGE+");")
127                }
128            }
129
130           RS5 = SQL.EXE("SELECT * FROM BCM_TE");
131
132           While(RS5.NEXT())
133           {
134                 if (RS5.EN_NAME == "_"+BF_NAME+".LEAVE")
135                 {
136                     Print("Event ev = new Event("+RS5.EN_NAME+");");
137
138         Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
139   ime());");
140                 }
141
142           }
143               Print("return mt;\n");
144           &>
145         }
146
147
148       Public Material UnLoadMaterial(Integer idx)
149       {
150               Material mt = _material.unloadAt(idx);
151
152          <&
153            RS4 = SQL.EXE("SELECT * FROM CUM_DCI");
154
155            While(RS4.NEXT())
156            {
157                if(RS4.EVENTNAME == "BF_"+BF_NAME+".UnLoadMaterial")
158                {
159                  Print(BCInstanceName+".CU_"+RS4.CUM_NAME+
160                    ".UpdateInfo("+RS4.MESSAGE+");")
161                }
162            }
163
164           RS5 = SQL.EXE("SELECT * FROM BCM_TE");
165
166           While(RS5.NEXT())
167           {
168                 if (RS5.EN_NAME == "_"+BF_NAME+".LEAVE")
```

```
169                 {
170                     Print("Event ev = new Event("+RS5.EN_NAME+");");
171
172         Print("_"+BC_NAME+".so.ScheduleEvent(ev,_"+BC_NAME+".so.TNOW.getT
173   ime());");
174                 }
175
176           }
177               Print("return mt;\n");
178          &>
179         }
180
181         Public boolean IsBlocked()
182         {
183                if (_material != null)
184                { return true; }
185                else
186                { return false;}
187
188         }
189
190         Public Material GetMaterial(Integer idx)
191         {
192               return _material.getAt(idx);
193         }
194
195
196         Public Integer GetCapacity()
197         {
198               return _capacity;
199         }
200
201   }
```

Material_TP

```
1   Material_TP.Class Template
2
3   Public class Material
4   {
5         STRING _type;
6         STRING _status;
7         STRING _id;
8
9
10        Public String getType()
11        {
12              return _type;
13        }
14
15        Public void setType(String type)
16        {
17              _type = type;
18        }
19
20        Public String getStatus()
21        {
22              return _status;
23        }
24
25        Public void setStatus(String status)
26        {
27              _status = status;
28        }
29
30        Public String getId()
31        {
32              return _id;
33        }
34
35        Public void setId(String id)
36        {
37              _id = id;
38        }
39
40  }
```

A.3 Interfaces

BCInterface_TP

```
BCInterface_TP.Class Template

Public Interface BCInterface
{
      Public void ReadyToLoad(TSInterface ts, LocationInterface
destination);
      Public void ReadyToUnload(TSInterface ts, LocationInterface
source);
}
```

TSInterface_TP

```
TSInterface_TP.Class Template

Public Interface TSInterface
{
      Public void MoveOrder(Material mt, LocationInterface source,
LocationInterface  destination);
      Public void LoadMaterial(Material mt, LocationInterface source);
      Public Material UnloadMaterial(LocationInterface destination);

      Public void init(STRING BCName);
}
```

LocationInterface_TP

```
LocationInterface.Class Template

Public Interface LocationInterface
{
      Public void LoadMaterial(Material mt);
      Public Material UnloadMaterial();
      Public Boolean IsBlocked();
      Public Material GetMaterial();
}
```

REFERENCES

Adiga, S. and C.R. Glassey (1991). "Object-Oriented Simulation to Support Research in Manufacturing Systems," *International Journal of Production Research*, Vol. 29, No. 12, pp. 2529-2542.

Alexopoulos, C. and A. F. Seila (1998). "Output Data Analysis," *Handbook of Simulation* edited by J. Banks, John Willey & Sons Inc, New York, NJ, pp. 225-272.

Arango, G. and R. Prieto-Diaz (1991). "Domain Analysis Concepts and Research Directions," *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Los Alamitos, CA, pp. 9-33.

Balci, O. (1998). "Verification, Validation, and Testing," *Handbook of Simulation* edited by J. Banks, John Willey & Sons Inc, New York, NJ, pp. 335-396.

Banks, J. (1998). "Software for Simulation," *Handbook of Simulation* edited by J. Banks, John Willey & Sons Inc, New York, NJ, pp. 813-835.

Banks, J., J.S. Carson II, and B.L. Nelson (1996). *Discrete –Event System Simulation*, 2nd ed., Prentice Hall, Upper Saddle River, NJ.

Bodner, D.A. and L.F. McGinnis (2002). "A Structured Approach to Simulation Modeling of Manufacturing Systems," *Proceedings of the 2002 Industrial Engineering Research Conference*, Orlando, FL.

Brooks, R.J. and A.M. Tobias (1996). "Choosing the Best Model: Level of Detail, Complexity, and Model Performance," *Mathl. Conput. Modeling*, Vol. 24, No. 4, pp. 1-14.

Brooks, R.J. and A.M. Tobias (2000). "Simplification in the Simulation of Manufacturing Systems," *International Journal of Production Research*, Vol. 38, No. 5, pp. 1009-1027.

Budd, T.A. (2001). *Introduction to Object oriented Programming*, 3rd ed., Addison-Wesley Pub. Co.

Carriero, D. and N. Gelernter (1992). "Coordination Languages and Their Significance," *Communication ACM*, Vol 35, No 2, pp. 96-107.

Cassandras, C.G., S. Lafortune, and S. Safortune (1999). *Introduction to Discrete Event Systems*, Kluwer Academic Publishers.

Chandy, K.M., and J. Misra (1981). "Asynchronous Distribute Simulation via a Sequence of Parallel Computations," *Communication of the ACM*, Vol. 24, No, 4, pp. 198-205.

Easttom, C. (2001). *Advanced JavaScript*, 2nd ed., Wordware Publishing.

Frantz, F.K. (1995). "A Taxonomy of Model Abstraction Techniques," *Proceedings of the 1995 Winter Simulation Conference*, SCS, pp.1413-1420.

Fujimoto, R.M. (2000). *Parallel and Distributed Simulation Systems*, John Wiley & Sons Inc., New York, NJ.

Gross, D., D. Pace, S. Harmon, and W. Tucker (1999). "Why Fidelity?," *Spring 1999 Simulation Interoperability Workshop Paper 3*, pp. 1055-1061.

Gross, D.C. (1999). "Report from the Fidelity Implementation Study Group," 99S-SIW-167, *1999 Fall Simulation Interoperability Workshop Paper*, March 1999.

Innis, G.S. and E. Rexstad (1983). "Simulation Model Simplification Techniques," *Simulation*, Vol .40, No.1, pp. 7-15.

Jefferson, D.R., B. Beckman, et al. (1987). "The Time Warp Operating Systems," *The 11th Symposium on Operating Systems Principles*, Vol. 21, pp. 77-93.

Johnson, M. E., and M. Mollaghasemi (1994). "Simulation Input Data modeling," *Annals of Operations Research*, Vol. 53, pp. 47-75.

Kempf, K.G (1996). "Simulating Semiconductor Manufacturing Systems: Successes, Failures, and Deep Questions," *Proceedings of the 1996 Winter Simulation Conference*, IEEE, Piscataway, NJ, pp. 3-11.

Kim, H., C. Zhou, and Hua X. Du (2000). "Virtual Machines for Message Based Real-Time and Interactive Simulation," *Proceedings of the 2000 Winter Simulation Conference*, Vol. 2, IEEE, Piscataway, NJ, pp. 1529-1532.

Kim, H., J. Park, S. Sohn, Y. Wang, S. Reveliotis, C. Zhou, D. Bodner, and L. F. McGinnis (2001). "A High Fidelity, Web-based Simulator for 300mm Wafer Fabs," *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference*, Vol. 2, Piscataway, NJ, pp. 1288-1293.

Kim, H. S. Sohn, Y. Wang, T. Tezcan, L. F. McGinnis, and C. Zhou (2002). "A Simulation Architecture with Distributed Controllers for Cell-Based Manufacturing Systems," *Proceedings of the 2002 Winter Simulation Conference*, Vol. 2, IEEE, Piscataway, NJ, pp. 1995-2002.

Kleijnen, J.P.C. (1998). "Experimental Design for Sensitivity Analysis, Optimization, and Validation of Simulation Models," *Handbook of Simulation* edited by J. Banks, John Willey & Sons, Inc, New York, NJ, pp. 173-224.

Kleijnen, J.P.C. (1975). *Statistical Techniques in Simulation*, Vol 2, Marcel Dekker, NY.

Kline, K. (2000). *SQL in a Nutshell: A Desktop Quick Reference*, O'Reilly & Associates

Klir, G. J. (1985). *Architecture of systems Complexity*, Sauders, NY.

Law, A. M, and D. W. Kelton (1999). *Simulation Modeling and Analysis*, 3rd ed., McGraw-Hill, New York, NJ.

Law, A.M., and M. G. McComas (1998). "Simulation of Manufacturing Systems," *Proceedings of the 1998 Winter Simulation Conference*, pp. 49-52.

Malone, T.W. and K. Crowston (1994). "The Interdisciplinary study of coordination," *ACM Computing Surveys*, 1994, Vol. 26, No 1, pp. 87-119.

Mize, J.H and D.B. Pratt (1991). "A Comprehensive Object-Oriented Modeling Environment for Manufacturing systems," *Proceedings of 2nd Industrial Engineering Research Conference*, IIE, Norcross, GA, pp. 700-704.

Narayanan, S. (1994). *Design and Development of an Object-Oriented Architecture for Modeling and Simulation of Discrete Part Manufacturing Systems*, PhD Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA.

Neter, J. M. H. Kutner, C. J. Nachtsheim, and W. Wasserman (1990). *Applied Linear Statistical Models*, 4th ed. IRWIN, Chicago, IL.

Overstreet, C. M. (1982). *Model Specification and Analysis for Discrete Event Simulation*, PhD Dissertation, Department of Computer Science, Virginia Tech, Blackburg, VA.

Overstreet, C.M., and R. E. Nance (2002). "Issues in Enhancing Model Reuse," *Proceedings of First International Conference on Grand Challenges for Modeling and Simulation* held as part of 2002 SCS Western Mutilconference.

Pace, D. K. (1998) "Dimensions and Attributes of Simulation Fidelity," *1998 Fall Simulation Interoperability Workshop Paper*, September 1998.

Paul, R.J. and S. J. E. Taylor (2002). "What Use Is Model Reuse: Is There A Crook at the End of Rainbow?," *Proceedings of the 2002 Winter Simulation Conference*, IEEE, Piscataway, NJ, pp. 648-652.

Pegden, C.D, R.E. Shannon, and R.P. Sadowiski (1990). *Introduction to Simulation Using SIMAN*, Mc-Graw-Hill Inc., Princeton Road, NJ.

Perumalla, K. S. and R. M. Fujimoto (1998) "Efficient Large-Scale Process-Oriented Parallel Simulations," *Proceedings of the 1998 Winter Simulation Conference*, IEEE, Los Alamoitos, CA, pp. 459-466.

Roza, M, P.V. Gool, and H. Jense (1998). "A Fidelity Management Process Overlay onto the FEDEP Model," *Fall 1998 Simulation Interoperability Workshop Paper*, (98F-SIW-083).

Sargent, R. G (1988). "Event Graph Modeling for Simulation with an Application to Flexible Manufacturing Systems," *Management Science*, Vol. 34, No. 10, pp. 1231-1251.

Sargent, R. G (1996). "Verifying and Validating Simulation models," *Proceedings of the 1996 Winter Simulation Conference*, IEEE, Piscataway, NJ, pp. 55-64.

Schricker, B.C., R.W. Franceschini, and T.C. Johnson (2001), "Fidelity Evaluation Framework," *Proceedings of 34th Annual Simulation Symposium*, pp. 109-116.

Schruben, L. (1983). "Simulation Modeling with Event Graph," *Communication ACM*, Vol. 26, No.11, pp. 957-963.

Sevinc, S. (1991). "Theory of Discrete Event Model Abstraction," *Proceedings of the 1991 Winter Simulation Conference*, IEEE, Piscataway, NJ, pp. 1115-1119.

Sommerville, I. and M. Ramachandran (1991). "Reuse Assessment," *Proceedings of First International Workshop on Software Reuse*, Dortmund, German.

Stevens, P. and R. Pooley (2000). *Using UML Software Engineering with Objects and Components*, Addison-Wesley.

Travers, L. and S. Sevinc (1992). "Abstracting and Explaining Simulation Model Behaviour," *Proceedings of the 3rd Annual Conference of AI, Simulation, and Planning in High Autonomy Systems*, pp. 156-160.

Vencent, S. (1998). "Input Data Analysis," *Handbook of Simulation* edited by J. Banks, John Willey & Sons Inc, New York, NJ, pp. 55-92.

Willemain, T.R. (1994). "Insights on Modeling from a Dozen Experts," *Operations Research*, Vol. 42, No. 2, pp. 213-222.

Willemain, T.R. (1995). "Model Formulation: What Experts Think About and When," *Operations Research*, Vol. 43, No. 6, pp. 916-932.

Yücesan, E. and L. Schruben (1992). "Structural and Behavioral Equivalence of Simulation Models," *ACM Transactions on Modeling and Computer Simulation*, Vol. 2, No. 1, pp. 82-103.

Zeigler, B.P. (1984). *Multifaceted Modeling and Discrete Event Simulation*, Academic Press, London, UK.

Zeigler, B.P. (1991)."Object Oriented Modeling and Discrete Event Simulation," *Advances in Computers*, Vol. 33, pp. 67-114.

Zeigler, B.P. (1998). "Review of Theory in Model Abstraction," *Proceedings of SPIE on Enabling Technology for Simulation Science II*, Vol. 3369, pp. 2-13.

Zeigler, B.P., H Praehofer, and T.G. Kim (2000). *Theory of Modeling and Simulation*, 2nd ed. Academic Press, San Diego, CA.

VITA

Hansoo Kim was born in Seoul, Korea, on July 18, 1968. He received his B.S. degree in Industrial Engineering from Hanyang University, Seoul, Korea, and M.S. degrees in Industrial Engineering from Korea Advanced Institute of Science and Technology (KAIST), Taejun, Korea in 1991 and 1993 respectively.

From 1993 to 1998, he worked for Samsung SDS Co., Korea as a technical consultant for simulation modeling and analysis in the Computer Integrated Manufacturing (CIM) department, and as a senior researcher at the Open Solution Center in Information Technology R&D Center. In fall 1998, he started the Ph.D. program in the School of Industrial and Systems Engineering at the Georgia Institute of Technology, Atlanta, GA.

Since spring in 1999, he has joined the Keck Virtual Factory Laboratory as a project leader for HiFiVE (High Fidelity Virtual Environment for Manufacturing Systems). His research interests include development of methodology using simulation, optimization, stochastic process, and information technology for modeling, design, and analysis of manufacturing and logistics systems, as well as the associated software development.